AFIT/GEO/ENG/93M-02

FREQUENCY DOMAIN SPEECH COMPRESSION USING THE KARHUNEN-LOEVE TRANSFORM

THESIS

Donald W G Dryley

Flight Lieutenant, RAAF

Approved for public release; distribution unlimited

AFIT/GEO/ENG/93M-02

FREQUENCY DOMAIN SPEECH COMPRESSION USING THE KARHUNEN-LOEVE TRANSFORM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the Requirements for

the Degree of Master of Science in Electrical Engineering

Donald W. G. Dryley BEng(Hons)

Flight Lieutenant, RAAF

March 1993

ACKNOWLEDGEMENTS

The people who helped me with this study are numerous, some more conspicuous than others. Major Steve Rogers provided guidance and encouragement as my thesis advisor, while Captain Dennis Ruck's contributions with the NeXT workstation and associated software were essential for the completion of the study. I am also grateful for the assistance with the NeXT Sound format and C programming language provided by Captain Jim Geurts.

ABSTRACT

The purpose of this study was test the influence of phase on the quality of speech reproduced by a speaker dependent compression system. The tests consisted of compressing frequency domain speech vectors using the Karhunen-Loeve Transform, with and without phase, then making subjective judgements as to the reproduced quality. Error Metrics were then tested for their suitability as predictors of reproduced quality.

The compression software transformed each speech vector into a vector of complex Fourier coefficients (only half of the coefficients are needed as transform is hermitian). Phase was preserved by using the real frequency components to form one vector and the corresponding imaginary components to form a second vector of real numbers which were then separately compressed. The expanded vectors were recombined and speech reconstructed by Inverse Fourier Transformation.

Compression ratios of 8:1 could be achieved without any perceivable difference between the original speech and reconstructed speech by minimizing the MSE of each vector of the pair. The 8:1 Compression Ratio corresponded to a covariance matrix Condition Number of 200.

Recommendations for further study into voice characterization and an optimal transform for speech are made.

TABLE OF CONTENTS

INTRODUCTION	1
Speech Compression	2
Primary Issues	3
Intelligibility	3
Quality	3
Metrics	4
Modelling	4
Speech Characterization	4
Proposal	6
Scope	7
Assumptions	7
Voice Characterization by Phonemes	7
Fourier Transform	8
Applicability of The Karhunen-Loeve Transform	8
Implementation	8
Quality Levels	9
Influence of Phase	9
Error Metrics	10
Summary	10
Subsequent Chapters	11
LITERATURE SEARCH	12
Linear Prediction Coding	13
Frequency Domain Compression	16
Previous Work at AFIT	16

Pols 1971 17

Karhunen-Loeve Transform 20

Chen and Huo 1991 21

Wavelet Transform 24

Training Set Design 25

Summary 27

EXPERIMENTATION 28

Overview 29

Equipment Used 30

Influence of Phase 30

Without Compression 30

With Compression 31

Error Metrics 33

Relative Mean Square Error 35

RMSE Per Coefficient 35

Condition Number 36

Voice Characterization 36

Suitability of The DFT and KL Transforms 37

DISCUSSION OF RESULTS 38

Phase and Quality 39

Phase Removal 39

Compression 41

Magnitudes Only 41

Phase Preserved 43

Error Metrics 44

Mean Square Error 44

Relative Mean Square Error 45

Relative Mean Square Error per Coefficient 46

Condition Number 47

Voice Characterization 48

Single Speaker 48

Multiple Speakers 49

Summary 50

CONCLUSIONS AND RECOMMENDATIONS 51

Influence of Phase on Quality 52

Error Metrics 53

Voice Characterization 54

Optimum Speech Transform 54

Recommendations 55

Applications 55

Further Study 56

BIBLIOGRAPHY 57

APPENDICES

Phase Removal Simulation A-2

Phase Removal with Compression Simulation B-2

Average Matrices Construction Program C-2

Covariance Matrices Construction Program C-8

Karhunen-Loeve Transformation Program C-13

Speech Reduction Program C-16

Recipes.h	C-23
Recipes.c	C-24
Data File Conversion Program	C-40
Sound File Conversion Program	C-43
Mean Square Error Program	D-2
Relative Mean Square Error Program	E-2
Relative Mean Square Error per Coefficient Program	F-2
Condition Number Program	G-2
Training Set	H-2
Testing Set	H-3

LIST OF FIGURES

1.	Original Speech	39
2.	Reproduced Speech, Magnitudes Only	40
3.	Speech used for Compression Tests	41
4.	Magnitudes Only, No Compression	42
5.	Magnitudes Only, 2:1 Compression	42
6.	Phase Preserved, 8:1 Compression	43
7.	MSE versus Number of KL Coefficients	45
8.	RMSE versus Number of KL Coefficients	46
9.	Condition Number versus Number of KL Coefficients	47

CHAPTER ONE

INTRODUCTION

Speech Compression

There are a number of signal processing techniques applied to speech signals which are associated with the term 'compression'. One common compression technique is to use a non-linear quantization on analog signals so that better resolution of low amplitudes can be achieved[1]. Another compression technique is to reduce the bandwidth of speech so that less channel bandwidth is needed to transmit the intelligence[2]. In both cases the signals are compressed, the first in amplitude and the second in frequency. This thesis is concerned with the frequency compression of speech and all further references to speech compression should be understood as such.

The reasons for speech compression are many. Speech signals that have been pre-processed to reduce their bandwidth are simpler signals as the redundant material has been removed. One advantage of this simplification is that applications such as Automatic Speech Recognition (ASR) can be implemented using a system that is less elaborate than that required if the compression step was neglected[3].

Communications is the application which has motivated most speech compression research[4]. Compressed speech requires less bandwidth than full bandwidth speech and so increases the throughput of a communication networks by allowing more channels to be allocated to a frequency band. The military uses speech compression to increase the throughput of its communication systems (in particular those networks connecting mobile nodes) and as a means of implementing narrow band secure voice systems[4].

Primary Issues

Regardless of the compression ratio (compression ratio is ratio of the original bandwidth to the bandwidth of the compressed speech), the performance of a compression system is ultimately decided by the intelligibility and quality of the reconstructed speech (note that this is not to say that compression ratio is unimportant).

Intelligibility

Intelligibility is a term used to describe how well the meaning of communication (intelligibility) is passed. It is perhaps most easily explained in terms of the talker (originator) asking the listener to repeat back the message. If the original message matches that sent back then there was high intelligibility. Intelligibility does not suggest the need to recognize the talker by their voice, only what was said.

Quality

Initially, let the quality of speech be defined as the degree to which a listener recognizes the talker's voice. The definition includes the ability to understand the intelligence of the message and the ability to at least hear, if not recognize, the sounds which characterize the talker. Therefore, the intelligibility aspects of speech are considered a subset of the quality aspects of speech. This definition is somewhat naive in that the listener may not know the talker and so could not be expected to recognize them by their voice over a communications channel. A more objective definition is that of *toll quality* which is the level of quality required to make the reconstructed speech indistinguishable from the bandlimited

speech[4].

Metrics

Many authors provide definitions of intelligibility and quality. The above definitions are non-specific in that there are not any objective measurements that can be applied. Intelligibility could be measured by using some sort of word error rate metric but, because of human intuition, whole words could be misunderstood and yet the intelligence still conveyed accurately. However, a word error rate metric would be an objective measure of intelligibility. The definition for toll quality is succinct and easily understood; however, no indication is given to "the level of quality" or how this level is measured. It is then important to decide some metric by which compression ratios and overall system performance can be measured so that application constraints, such as available channel bandwidth and the need to understand the communication, can be used for design criteria.

Modelling

Perhaps the most important issue is that of modelling human speech and hearing. A variety of speech compression models exist, the most popular of which is the speaker independent Linear Predictive Coding (LPC) model[4]. Modelling, in general, will be further discussed in the following section on speech characterization.

Speech Characterization

Modelling is a standard and valid scientific approach to characterizing any process. A parametric speech model is most desirable as

varying the parameters produces different sounds and so speech, consisting of a range of sounds, can be represented by different sets of model parameters. If the parameters of the model can be represented by a lower number of quantities than the speech sample being modelled, then compression has been achieved.

It is typical for the model to only approximate the original speech which leads to a reconstruction error. Expectations are that the lower the reconstruction error, the better the quality of the reconstruction (assuming a suitable metric is used). However, speech is a highly redundant signal for conveying intelligence and large reconstruction errors can be tolerated without reducing intelligibility. If intelligibility is the most important performance criteria, then a speaker independent compression model such as LPC would be useful.

The LPC model, or its variants, is used effectively in speech compression applications as it provides intelligible, speaker independent, reproduction of compressed speech. A typical compression ratio is that of the STU-III which compresses 64 Kbps to 2400 bps[4]. The major problem with LPC based systems is that of noise[4] (noisy signals corrupt LPC model). Further, the quality of the reproduction is reported as poor because speakers cannot be identified[4].

When the quality of reproduced speech is discussed, most often the results of communications applications are presented (as in the example above). But speech compression is also used in speech recognition applications which generally do not require the quality aspects of speech to

be preserved. Consequently, the speaker independence of the LPC model makes it useful for speaker independent speech recognition applications (an LPC model is used to compress utterances to the vocal tract model thereby reducing the dimensionality of the feature space to the number of model parameters). However, if *speaker verification*[4] is viewed as a recognition task with the additional requirement of preserving the quality aspects that characterize a voice, then the LPC compression model is not satisfactory. Because of this need to preserve the quality of speech, a compression model which supports speaker verification should also be applicable to communication applications.

One of the principles of recognition is to learn a speaker's voice patterns using labelled data (i.e. get a speaker to say pre-determined words) and then recognize subsequent voice patterns. Studying voice patterns may provide some insight into the compression model capable of quality speech reproduction.

Proposal

It is believed that the quality of a speech compression system such as LPC can be improved if the speaker independence is traded for speaker dependence. Therefore, a speaker-dependent frequency domain speech compression system is proposed. Users will first train the system by generating a training set from which the principal components of the speech will be derived. Subsequent speech inputs will be mapped into the compressed space spanned by the principal components and transmitted. At the receiver, the compressed space is mapped back to the dimensions of the original space and the speech reproduced.

The training set will consist of frequency vectors which are the Fourier Transform (FT) of discrete-time samples of a user's voice. The frequency vectors will form a covariance matrix of which the eigenvectors will form the rows of a Karhunen-Loeve (KL) Transform matrix. When used, the frequency vectors will be multiplied by the KL transform matrix to produce a set of KL coefficients. The resulting set of coefficients will be transmitted through a narrow-band channel.

Prior to transmission, the KL transform matrix will be transmitted to the receiver where it will be inverted and used to inverse transform the set of KL coefficients upon reception. Note, that the KL transform matrix is made up of orthogonal vectors; consequently, matrix inversion can be achieved by transposing the matrix.

Scope

This thesis will use the Karhunen-Loeve transformation for speech compression, examining: the effect of preserving phase and error metrics for quantifying reproduced quality. The results will also be used to conclude the effectiveness of characterizing a voice by a set of phonemes.

Assumptions

Voice Characterization by Phonemes

The first of the assumptions is that a speaker can be properly characterized by learning the range of sounds that he, or she, makes. The system proposed builds a model of the user's voice from a set of phonemes.

The phoneme set is represented by sentences which consist of all possible phoneme bigrams (i.e. 40 phonemes has 40^2 bigrams). Once a comprehensive sample of the user's voice is obtained, the goal becomes one of extracting the information that describes quality (the foundation of the proposal).

Suitability of The Fourier Transform

The second assumption is that the linear frequency scale of the FT will be suitable to represent speech. The doubt results from the linear frequency scale of the FT versus the non-linear frequency scale of human perception.

Applicability of The Karhunen-Loeve Transform

It is also assumed that the KL transform will yield a compression model that concisely represents the quality aspects of speech. This assumption is based upon the optimality criteria of the KL transform which, essentially, selects the most important frequencies for transmission by minimizing the mean-square-error (MSE) between the original and reproduced frequency vectors. By minimizing MSE it is expected that the quality aspects of speech will be preserved.

Implementation

The proposed method of implementation is to use a parallel structure to make vector multiplications. An ideal structure for this is the Artificial Neural Network (ANN) which has the input frequency vector presented to the layer of input nodes and the eigenvectors of the KL transform are the weights connecting the input layer to the hidden layer.

Quality Levels

Before the results of any tests can be compared, some sort of measurement standards need to be established. Speech quality is highly subjective as human perception is involved. In particular, if a listener is familiar with the speaker's voice then making impartial decisions on quality is difficult. Consequently, some 'quality levels' need to be defined.

Excellent Quality. Excellent quality is achieved when the reproduction is indistinguishable from the original.

Good Quality. Good quality is achieved when the speaker is recognizable (i.e. the signal may be noisier than the original but without distorting the speech).

Intelligible. Intelligible reproduction is speech that can be understood without the listener having to interpolate confusing or badly distorted words.

Unintelligible. Unintelligible speech is defined here as speech which cannot be clearly understood.

Influence of Phase

There is no known reason for assuming that phase will affect reproduction quality. Speech recognizors do not require phase information; however, it is wrong to assume phase plays no role in quality. There are

many non-speech examples of the importance of phase information, optics contains many and it is certain that other disciplines do also.

Error Metrics

The MSE is often used as an error metric. However, if the phase spectrum is distorted in such a way to maintain magnitude, then it is expected that the reproduced quality will decrease even though the MSE may in fact decrease. What is needed is a way of measuring the relationship between the frequencies of a speech vector.

Summary

Speech is compressed to reduce the bandwidth requirements of voice communications and as a pre-processing stage to speech recognition. Intelligibility is the criteria driving narrow-band secure voice communications and is implied in the performance of speech recognition systems. However, speaker verification applications require reduced bandwidth while maintaining the quality aspects of speech.

A speaker-dependent speech compression system which maintains the quality aspects of the user's voice is proposed. The system will model the user's voice by 'learning' a 'training set' of naturally spoken phonemes then compress subsequent utterances using the characterization model. If used for communications, the speech will be reconstructed from the compressed data transmitted. The goal is to achieve toll quality.

The scope of the study is to firstly test the influence of phase on quality and the suitability of some error metrics used to measure quality. The assumptions concerning the characterization of a voice with a set of phonemes and the preservation of quality with the KL transform will be discussed.

Subsequent Chapters

Chapter two contains a search of relevant literature and discussions on topics raised. The actual experiments performed are detailed in chapter three. Experimental results are discussed in chapter four while chapter five contains conclusions and recommendations for further research. Software developed for the research is listed in the appendices.

CHAPTER TWO

LITERATURE SEARCH

Weinstein's review of military speech processing applications[4] revealed that the LPC algorithm is the most popular of all compression techniques used for communications. Parson's[5] points out the versatility of LPC by describing the techniques use as a pre-processing stage for speech recognition systems. Because LPC is popular it is worth investigating how it works. Because the method has deficiencies, being intolerance to noise and poor reproduction quality[4], understanding the principles of LPC might provide some insight into the broader compression problems.

Linear Prediction Coding

Linear Predictive Coding is an application of adaptive linear prediction. Strobach[6] writes, "The linear prediction model provides a parametric description of an observed process." He continues with the development, "The idea of parametric process modelling in context with linear prediction methods is based on the assumption that a signal is completely determined in terms of its first-order (mean) and second-order (covariance) information when only the parameters and the excitation of the generating filter (process model) are known." Essentially, the next value associated with an observation can be predicted if: a sequence of values associated with the preceding observations are known; and, if the statistics of the process under observation are stationary. The length of the sequence of preceding values is proportional to the accuracy of the prediction (i.e. many preceding values, high accuracy). Using this technique, a set of previous discrete-time measurements (set of frames of samples) can be used to generate a speech model that predicts the current measurement (frame of samples) and the current

measurement is used to update the speech model for the subsequent prediction.

The speech model is derived from the physiology of speech production. The vocal tract is a flexible hollow muscle which can be deformed by the speaker to produce different sounds[7]. Excitation for the vocal tract can be air modulated by the vibrating vocal chords (voiced sounds) or air passing through the rigid (non-vibrating) vocal chords (unvoiced). The LPC model samples the utterance over a period of 20 - 50 milliseconds (assumes stationarity) and models the vocal tract configuration by cylinders of differing cross-sectional-areas. The excitation is modelled as either: noise for unvoiced sounds; or, impulses at the same pitch as the vibrating vocal chords (voiced sounds). The excitation and vocal tract model is transmitted to a listener where the utterance is reconstructed by exciting the model vocal tract.

The popularity of LPC is derived from its reproduction of intelligibility which, for communications, is the essential performance criterion. However, for speaker verification the quality aspects of speech need to be carried with the model and LPC does not provide this feature[4]. Where do the quality aspects of speech lie?

Makhoul[8] refers to the quality aspects of LPC in terms of the accuracy of reproducing the predictor parameters, "It has been known for some time that the quantization of the predictor parameters themselves is quite inefficient since a large number of bits is required to retain the desired

fidelity in the reconstructed signal at the receiver[72][1]." This statement appears to be suggesting that a quality reproduction is possible using linear prediction, but the overhead involved is prohibitive.

Makhoul[8] also defines his error metric, E, in terms of the ratio of original and reproduced power spectral envelopes.

$$E = \frac{1}{2\pi}\int_{-\pi}^{\pi} [\log \frac{P(\omega)}{\bar{P}(\omega)}]^2 \, d\omega \qquad (1)$$

where P is the original spectral envelope,
$\bar{P}$ is the reconstructed spectral envelope

Notice that there is no way of including phase errors between the two spectrums using this error metric. Does the phase spectrum matter? White[9] writes, "It is generally believed that all acoustic information relevant to speech recognition is represented by the time evolution of the power spectrum, with the phase component being relatively unimportant." White's comments are related solely to speech recognition which does not require reconstruction and so phase may not be important. However, for applications that require *quality* reconstructions phase may be important. Perhaps frequency domain coding can provide more information regarding the importance of phase.

---

[1] Makhoul's reference

Frequency Domain Compression

A frame of speech samples can be considered a vector of dimension n, where n is the number of samples in the frame. The discrete-time samples are real and so the Fourier Transform (FT) is Hermitian[10]. The FT is completely invertible and, by Parseval's theorem, energy is conserved[11]. Therefore, the frame of n discrete-time speech samples can be represented by complex coefficients at n/2 frequencies. The question to be answered is - Why transform to the frequency domain?

One reason is that the DFT (Discrete Fourier Transform) produces a set of measurements describing the spectral content of the particular frame of speech. These measurements are often viewed as a set of cross-correlations between the discrete-time speech and a complex frequency set[12]. For some, the DFT is more intuitive than the discrete-time representation as it is a set of measurements of 'content'.

Another reason for investigating frequency domain compression models is that AFIT has done some research in this area recently[13].

Previous Work at AFIT

The most recent speech compression work done at AFIT was completed in December 1991 by Captain Shane Switzer[13]. Captain Switzer investigated methods for better coding fricatives and plosives. Parsons[5] description of fricatives includes the use of the characteristic frequency spectrum which is used to locate constrictions of the vocal tract. If the vocal tract is

completely shut off for a short time then quickly released, the sharp sound is called a stop or plosive. Captain Switzer's research revealed that shorter sampling windows were better for coding fricatives and plosives. This result is reasonable as the statistics of such signals varies quickly and short windows should be better for maintaining stationarity.

A point of interest regarding Captain Switzer's compression technique was his arbitrary selection of frequencies for transmission. He tried a number of selection methods, one where he simply selected the frequencies of greatest magnitude. The result of this selection technique was an intelligible reproduction with a compression ratio of 64000/2400 (equivalent of the STU-III using the LPC model[4]). Can reproduced quality be improved by a better selection of frequencies? If so, how are these 'better' frequencies selected.

Pols 1971

Louis Pols[14] applied 15 ms speech frames, bandlimited to 8 KHz, to a bank of 20 1/3 octave filters for spectral analysis. He combined the outputs out of lowest three filters, and the next lowest two filter outputs were similarly configured, to produce 17 spectral components. The 17 frequency components are analysis using Principal Component Analysis (PCA). There are two steps to PCA: firstly, a 17 x 17 covariance matrix is formed from the 15 ms frames; and, then the covariance matrix is analyzed for the directions of maximum variance.

Pols[14] experimented with 20 speakers speaking the same 20 Dutch

words. His results showed that the variance was greatest in the high frequency regions where the vowels, nasals and fricatives varied the most. Applying PCA he was able to reduce the 17 dimensional speech frames to 3 dimensions. Analysis of these dimensions showed that they represent 78.1% of the energy and that no other dimension has more than 5.6% of the energy. Dimension one (49.1%) appeared to discriminate between sonorants and non-sonorants since it separates the low and high frequency ends of the spectrum (sonorants are voiced sounds, i.e the vocal tract excitation is modulated by the vibrating vocal chords). Dimensions two and three had their highest weightings in the spectral regions of the formant frequencies.

Speech frames were then mapped into the principal components space and, after some time normalization, compared to the 20 reference words. 395 out of 400 words were correctly classified.

The results of these experiments need to be put into perspective. Firstly, while PCA is a powerful tool for analyzing and compressing a feature space, and Pols has shown that the compressed space is suitable for speech recognition, the same speakers and words were used for both training and testing (i.e. the actual utterances used to characterize the feature space were different examples of the same utterances used for testing the system).

Secondly, Pols' results give no indication of which speaker spoke the word that was classified.

Thirdly, Pols makes an interesting comment when discussing time normalization. "One possible correct time normalization, apart from a non-

linear approach[14][2], is to treat the individual sounds within an utterance separately, taking into account the target positions (often not actually reached) of short sounds in a context." This comment will be addressed later in the chapter.

Lastly, the type of spectral analysis used is interesting in that 1/3 octave filters provided a constant bandwidth to center-frequency ratio ($\Delta f/f_o$) and covered the band to 8 KHz without any phase information. This constant-Q filter arrangement will be discussed in relation to the suitability of the Fourier Transform later in the chapter.

Pols' paper shows that if the speech to be recognized is correctly characterized, then the frequency domain is a suitable space in which to perform recognition. Also, some of the energy can be removed (in this case 21.9% is unnecessary) by only extracting the principal components of a speech sample.

Analysis of the training set to find the directions of maximum variance and then using these directions of most significance to describe a compressed space could be a technique for the finding the important frequencies to transmit. The Karhunen-Loeve (KL) Transform can also be used to find the directions of maximum variance.

---

[2] Pols reference.

Karhunen-Loeve Transform

When working with discrete data the KL transform is best understood from the transform matrix where the rows are the eigenvectors of the covariance matrix of the training set. The eigenvectors selected for the transform matrix are those eigenvectors associated with the eigenvalues of the covariance matrix of greatest magnitude. Subsequent speech vectors are multiplied by the transform matrix such that a vector of KL coefficients results, the number of which are determined by the number of rows used in the KL transform matrix. That is:

$$\underline{Cov} = (\boldsymbol{x}_i - \bar{\boldsymbol{x}})\,(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T \tag{2}$$

where $\boldsymbol{x}_i$ *is the ith input column vector of the training set*

$\bar{\boldsymbol{x}}$ *is the vector of component averages*

T denotes transpose

$$\overline{x_k} = \frac{1}{N}\sum_{j=1}^{N} x_{jk} \tag{3}$$

where k is the kth frequency component of $\bar{\boldsymbol{x}}$*, and*

N is the number of vectors in the characterization set.

$$\text{Let } \boldsymbol{r}_j = \boldsymbol{eig}(\underline{Cov}) \tag{4}$$

be an eigenvector that is associated with the jth eigenvalue of $\underline{Cov}$.

These vectors form an orthogonal 'eigen-space' into which the KL transform maps subsequent speech vectors. The inverse transform results from

$$KLC = \mathbf{T}\,(\mathbf{x}-\bar{\mathbf{x}})$$

where KLC *is a vector of n coefficients,* $\mathbf{T}$ *is the KL transform matrix with n rows,* $\mathbf{x}$ *is an input vector, and* $\bar{\mathbf{x}}$ *is the vector of means*

(5)

the vector of KL coefficients being multiplied by the transpose of the KL transform matrix. The KL transform has the optimal properties of: minimizing mean-square error when a reduced number of coefficients are used; and, minimizing the entropy function defined in terms of the average squared coefficients used[15].

Chen and Huo 1991

A recent use of the KL transformation for speech compression was that of Chen and Huo[16]. In this study, they reported, "... up to 70% data reduction is possible with no appreciable degradation of the signal." A point of interest is the way they represented their speech. which was as vectors of Fourier-Bessel (FB) coefficients. The reason for using the FB Transform was based upon Rabiner and Schafer's[17] linear speech prediction model where the glottal pulses have the form of a freely decaying oscillation. "The Bessel function also displays amplitude-decaying and nonuniform zero crossing characteristics. The use of the Bessel function as the basis function for speech signal decomposition therefore seems logical and natural."[16]

Chen and Huo[16] performed a number of experiments and reported their results in terms of reproduction quality and compression ratio. Two of

these results were: for 150 Fourier-Bessel (FB) coefficients *excellent* quality was achieved with 10 KL coefficients (99.3% of the energy); and, for 80 FB coefficients *good* quality was achieved with 10 coefficients (97.6% of the energy).

There are at least three important aspects to the experiments and results of Chen and Huo[16]. They chose the FB transform as they felt it was better model for speech production. Qualitative terms such as 'excellent' and 'good' were used without any specific definitions to allow repeatability (note that there is no doubt that they achieved excellent and good quality). The KL transform was successfully used for speech compression.

The terms excellent, good, bad, etc need to be defined so that some sort of repeatable datum can be established for design and verification purposes. What sort of error metric will best match the reproduced quality definitions?

The FB transform may appear to be a better transform; however, Chen and Huo report that 70% percent of transform's end product was shown to be redundant. What sort of results should be expected from the best transform? Can KL give an indication as which transformation, if any, is best?

Fourier Transform

The DFT is a commonly used technique for representing discrete-time signals because it is completely invertible and makes otherwise difficult

mathematical operations (convolution and correlation) simpler. The ability to convolve and correlate signals using the FT makes it a powerful signal processing technique. The DFT also satisfies intuition as it provides a measure of a signal's spectral content by correlating the signal with a set of complex sinusoids[12]. Consequently, intuition and familiarity make the DFT a comfortable, and powerful, spectral analysis tool. Given that the DFT is completely invertible for discrete-time applications and preserves the phase information of a signal, it is logical to use it to represent such signals.

The DFT can be viewed as representing signals on a linear complex frequency axis. But it is well known that human perception is non-linear. Parsons[5] describes the MEL scale which approximates the response of human hearing as being linear below 1 KHz and logarithmic above 1 KHz. Then, it is also reasonable to suggest that speech should be represented by some non-linear transform which matches human perception rather than the linear FT.

Selecting the particular non-linearity is beyond the scope of this research. However, the DFT may provide some insight into the correct non-linearity. If a speaker's voice is correctly characterized then the vector of component averages defined in equation three will describe the average frequency response of the speakers voice signals. This information may lead to a suitable transform. The non-linear transform selected needs to be completely invertible and so the Discrete Wavelet Transform (DWT) may be suitable.

Wavelet Transform

The Wavelet Transform (WT) should, in this case, be considered as an alternative to the FT. The WT offers a means for implementing the constant-Q filtering technique used by Pols'[14] and, due to the logarithmic frequency scaling, better models human perception (see MEL scale[5]).

Rather than produce a set of cross-correlations between the signal and a discrete set of complex sinusoids, as with the DFT, the DWT performs cross-correlations with scaled and shifted versions of the 'mother wavelet'[18]. The mother wavelet can be considered analogous to a DFT 'window' function except that the operation is one of correlation not convolution.

Selection of the mother wavelet would be the essential task for producing quality reproductions and, if properly chosen, the optimization of KL transform may not be necessary (assuming such a wavelet exists). The KL transform as described above requires the 'average spectrum' (equation three) of a speaker's voice to construct a covariance matrix and the eigenvectors of the covariance describe the directions of maximum variance. This information is derived from the set of vectors that characterize the speaker's voice and so may be useful for designing a mother wavelet.

Regardless of the representation of the speech samples, if a speaker dependent speech model is used then a means for characterizing the speakers voice is needed. This characterization will be known as training set design.

Training Set Design

A problem with the training sets used by Pols[14] and Chen and Huo[16] is that examples of the testing set were used to characterize the speakers' voices. Obviously this type of training set is not possible for a communications system as it suggests the speech must be known before it was spoken. What is needed is a means for characterizing a speaker's voice independent of the words actually spoken. A training set comprised of the range of sounds made would be independent of any words actually compressed. Pols[14] alluded to this when he referred to using "... sounds within an utterance..." as the reference for time normalization purposes.

White[9] also describes the role of phonemes in speech recognition, he writes, "Spoken words can be represented as strings of phonemes – the basic building blocks of speech. In spoken English[American][3] about 38 phonemes (16 vowels and 22 consonants) are typically used. There are two advantages to recognizing phonemes in a speech recognition system. Phonemes make possible (1) *selective recall of word prototypes* and (2) *reduction of memory requirements to store word prototypes* – i.e. data compression."

Pols[14] also made a comment regarding the approximation of the 'sounds which make up an utterance' when commenting on time normalization. It is difficult to extract Pols' contextual meaning of 'approximate'; however, it is known that the phonemes of a language are rarely clearly spoken[19].

---

[3] This Author's comment.

That is, speakers only approximate a phoneme then transit to making the next phoneme of the word. Consequently, words actually consist of approximated phonemes and the transitions. Is there a set of words containing all possible phonemes and transitions with which a training set can be developed to characterize speakers' voices?

Pattern recognition studies includes the study of training sets. A number of rules-of-thumb have been developed to choose the size of the training set with respect to the dimensionality of the feature space (Foley's Rule[20]) and with respect to the complexity of implementation (Widrow's Rules[20]). While these rules of thumb are based upon empirical studies, the underlying theme is the training set must represent the process being modelled. So when developing a training set to characterize the voice, all possible combinations of the sounds must be represented and these combinations must occur often enough to allow an accurate characterization. Assuming this training set exists, then how big should it be?

If the training set is based upon phonemes, then there are ≈40 phonemes and ≈1600 phoneme bigrams. So, the training set should include a number of examples of each of the 1600 phoneme bigrams.

An efficient training set design will result in a set of easily spoken sentences (capture speakers' natural way of speaking) with a uniformly distributed range of phonemes. There may be some minimization possible by adopting the principles of a grammar. That is, connected-word recognition can be improved by using the rules of grammar[4]. Grammar compilers that build training sets based on 'word' bigrams have been shown by Brown and

Wilpon[21] to improve connected word recognition. It may be that speakers never speak all 1600 phoneme bigrams and that some simplification of the phoneme based training set is possible by taking this into account.

Summary

It is not clear whether a compression model that preserves the phase spectrum of a speakers voice will support reproductions of better quality that those of LPC.

The DFT is an invertible measure of a discrete-time signal's complex frequency spectrum. The KL transform is an optimal compression technique (in a least squares error sense) that has been successfully used for speech compression. Consequently, the KL transform should support a compression model that preserves the phase information of the original speech.

The quality of speech tends to be a judgement rather than a measure and so some means of quantifying quality is needed if objective comparisons of compressions are to be made.

Due to the logarithmic frequency response of human hearing, the linear frequency scale of the FT may not be the best way of representing speech for compression.

The following chapter details the experiments that tested some of the assumptions made and associated issues raised in the proposal.

CHAPTER THREE

EXPERIMENTATION

Overview

The experiments use the KL transform to compress speech frequency vectors into KL coefficients. The two primary goals are to test whether preserving the phase components of the complex frequency vectors influences reproduced quality and to test the utility of four error metrics as predictors of reproduced quality.

The assumptions, that a voice can be characterized using a set of sounds that the speaker makes (phonemes) and that the DFT is a suitable representation of speech, will also be tested.

All quality assessments will be made according to those levels of quality defined in the proposal.

Before describing the experiments, a brief functional description of the software developed is warranted. The software developed for these experiments perform the following three functions: generate files containing the covariance matrix and a vector of component averages; decompose the covariance matrix into a KL transform matrix; and, transform discrete-time speech files to KL coefficients then reconstruct the speech by inverse transforming the coefficients.

The programs of Appendices B and C contains all of these functional modules; however, Appendix C has the modules spread over four listings and allows for the general case of a characterization set contained in multiple files. Also included in Appendix C are three other listings

which contain library functions and listings that convert sound files (NeXT format) to speech data files (integers), and vice versa. Specific tests have additional modules added and these will be referred to during the relevant discussions.

Equipment Used

The NeXT workstation was used as the speech processing platform. Speech was sampled at 11.02 KHz and 16-bit linear quantization using the NeXT's Sound Recorder software. Compression simulations were written in C and are attached in the Appendices.

Influence of Phase

This experiment will test the influence of phase on the quality of reproduced speech for two cases. The first case (listing of Appendix A) does not use compression and simply replaces the real and imaginary components of a speech vector with the magnitude (real) and zero (imaginary). The second case (listing of Appendix B) tests the influence of phase on reconstructed speech that has been compressed and then expanded.

Without Compression

The objective of the experiment is to establish whether preserving the phase of speech signals influences the quality of reconstructed spcech. The quality levels defined in the proposal (page 9) are used as references.

There are two steps to the testing process. Step one is to

simply listen to the unprocessed speech and evaluate it for quality. Step two calculates the magnitude of the complex frequency components and loads these values into the real locations of the Fourier transform input buffer. The imaginary locations are set to zero and the inverse Fourier transform performed. This discrete-time signal was replayed and evaluated for quality. The program of Appendix A was written for step two.

The above test provides an expectation benchmark for subsequent tests as well providing an indication of the influence of phase on quality. However, a more informative experiment is to test the influence of phase on the compression model. Consequently, the above comparison needs to be repeated with the compression and expansion simulation.

With Compression

Appendix B contains a listing that compresses the magnitude-only frequency coefficients. Compression is a more laborious task as a vector of component averages and a covariance matrix must first be determined. The covariance matrix is decomposed into singular values and eigenvectors and the KL transformation matrix constructed. Then the original speech can be compressed and expanded using the KL transformation. The phase preservation software of Appendix C is more complex only in that two sets of average, covariance, KL transform and KL coefficients are used to reproduce the real and imaginary components of the speech vectors.

Functional descriptions of the four listings at Appendix C are as follows. Listing one transforms the discrete-time samples of data files to 256-point frequency vectors and the vector of component averages saved to a

file along with the number of vectors used to determine these averages. Subsequent data files are then used to update this vector of averages.

Listing two determines a covariance matrix by re-reading the discrete-time data files, transforming to frequency vectors and subtracting the average spectrum from each frequency vector. The outer product of the difference between the input and average frequency vector is taken and added the covariance matrix (initialized to zero).

After all data files are read and the covariance matrix formed, the covariance matrix, A, is decomposed using the software of listing three into three matrices $V.W.V^T$. The columns of V are the eigenvectors of the covariance matrix. The non-zero elements of the diagonal matrix W contain the singular values (squareroots of the eigenvalues) which correspond to the eigenvectors of V. V^T is the transpose V and is saved as the KL transform matrix.

Listing four requests the number of coefficients used in compression model and this number is used to select the number of rows of the KL transform matrix. Frequency vectors are multiplied by the KL transform matrix to yield the vector of coefficients. The coefficients are then multiplied by the transpose of the KL transform to form the reconstructed frequency vector (note that the transpose of the KL transform matrix equals its inverse as the matrix is constructed from orthonormal eigenvectors).

Just as in the phase preservation experiment without compression, this experiment has two steps. The first step is to apply the KL transform

on the "magnitude only" frequency vectors and the second step preserves the phase of the speech vector. The listings of Appendix C preserve phase by treating each complex vector as a pair of vectors - one vector represents the real part of the frequency components and the other the imaginary parts. Then two vectors of averages, two covariance matrices, two KL transform matrices and two vectors of KL coefficients are required. The discrete-time reconstructions are replayed and their quality judged.

The advantage of representing a vector of complex numbers as dual vectors of real numbers is that half the computations are required to decompose two real covariance matrices as compared to one complex matrix[22].

Assuming that the KL transform provides optimal reconstructions of the two vectors, the relationship between the vectors (i.e. the phase) will be preserved to some optimal accuracy for the number of KL coefficients used.

Error Metrics

Four error metrics, Mean Square Error (MSE), Relative MSE (RMSE), RMSE per coefficient and Condition Number (CN) will be tested for their

utility in predicting reconstruction quality. MSE is defined in equation six.

$$MSE = \|E\|$$

$$\text{where} \quad E_i = \frac{1}{N} (REC_i - ORG_i)$$

is the ith component of the error vector, and

N is the number of vectors.

(6)

The RMSE is defined below in equation seven.

$$RMSE = \|E\|$$

$$\text{where} \quad E_i = \frac{1}{N} \frac{REC_i - ORG_i}{ORG_i}$$

is the ith component of the error vector, and

N is the number of vectors.

(7)

The RMSE per coefficient is the RMSE divided by the number of coefficients used for the reconstruction. The CN is defined below in equation eight.

$$CN = \frac{SV_1}{SV_x}$$

where SV_1 is the singular value with greatest magnitude **(8)**

SV_x is the xth singular value

Mean Square Error

The MSE measures the reconstruction error with respect to the original frequency vector. Consequently, listing four of Appendix C should be replaced by the listing of Appendix D to accumulate the relative error over the range of speech data transformed. At the end of the program the components of the error vector are averaged over the total number of frequency vectors transformed. The 2-norm magnitude of the error vector is then calculated as described by equation seven. Once the MSE is calculated, it is written to an error file. Note that, because complex frequency vectors are represented by two vectors of real numbers (one for the real and one for the imaginary parts of a frequency vector) there are two error spectrums and two RMSE measurements. Both are written to the error file and are later added to form a metric.

Relative Mean Square Error

The RMSE measures the reconstruction error with respect to the original frequency vector. Consequently, listing four of Appendix C should be replaced by the listing of Appendix E to accumulate the relative error over the range of speech data transformed. At the end of the program the components of the error vector are averaged over the total number of frequency vectors transformed. The 2-norm magnitude of the error vector is then calculated as described by equation seven. Once the RMSE is calculated, both measurements are written to an error file and are summed for a metric.

RMSE Per Coefficient

RMSE per Coefficient is the RMSE calculation divided by the number of coefficients used to generate the reconstructed vectors. By replacing

listing four of Appendix C with that of Appendix F the RMSE per coefficient will be written to the error file.

Condition Number

The positive squareroots of the eigenvalues of the covariance matrix are the singular values of the matrix[23]. The square of the eigenvalues are measurements of the energy represented by the respective KL coefficient[15] (inner product of the associated eigenvector and the frequency vector). Therefore, the ratio of the largest singular value to the nth singular value, or condition number for n singular values, is proportional to the energy represented by n KL coefficients. By determining condition number from the decomposition of the covariance matrix and associating it with quality levels (average of CN for both covariance matrices), a means for associating compression ratio and reproduced quality exists.

Appendix G lists the code for the CN measurements. The code that decomposes the covariance matrix and saves the KL transforms (listing three of appendix C) has been supplemented with code that calculates the CN and saves it to a file (i.e. Appendix G).

Voice Characterization

The assumption was that a voice could be characterized by a set of phoneme bigrams. In order to generate the 1600 (or so) phoneme bigrams, a comprehensive study of phonemics is required which is beyond the scope of this tnesis. However, the underlying principle can be tested by forming a compression model (KL transform matrix) from a training set of words

containing particular sounds and then compressing other words that can be constructed from these same particular sounds. For example, the words 'black' and 'clock' can represent the sounds /bl/, /lack/, /cl/ and /lock/ and so be used to characterize the word 'block'.

The training set is sampled and formed into discrete-time data files. The software of Appendix C is used to form the voice model and simulate the compression. The reconstructed data files are played back for a quality assessment and the number of coefficients used is noted. Appendix G contains a full list of training and testing data.

Suitability of The DFT and KL Transforms

The assumption that KL transforming the DFT is a suitable means for compressing speech should be examined. The reason for examining the DFT is based upon the disparity between the linear frequency scale of the DFT and the non-linear frequency nature of human hearing.

The eigenvectors of the covariance matrix provides the directions of greatest variance ranked according to the magnitude of the associated eigenvalues. The relationships described by the eigenvectors' directions may require analysis if any meaningful conclusions regarding the DFT can be made (beyond the scope of the thesis). However, prior to forming the covariance matrix an average spectrum is determined from the frequency vectors of the training set. If the speaker's voice is properly characterized, then analysis of the average spectrum may provide a starting point for determining an alternative to the DFT and/or KL transforms.

CHAPTER FOUR

DISCUSSION OF RESULTS

Phase and Quality

The influence of phase on reproduced quality was tested by comparing the results of two tests. The first test simply removed the phase components from the complex fourier coefficients and reproduced the discrete-time speech using the 'magnitudes only'. The reproduced speech was assessed for its quality.

The second experiment tests the influence of phase within the compression model. Vectors of fourier coefficients are transformed to sets of KL coefficients and then inverse-transformed back to frequency vectors. The experiment is performed for the 'magnitude only' and complex cases, after which the reproduced discrete-time speech is assessed.

Phase Removal

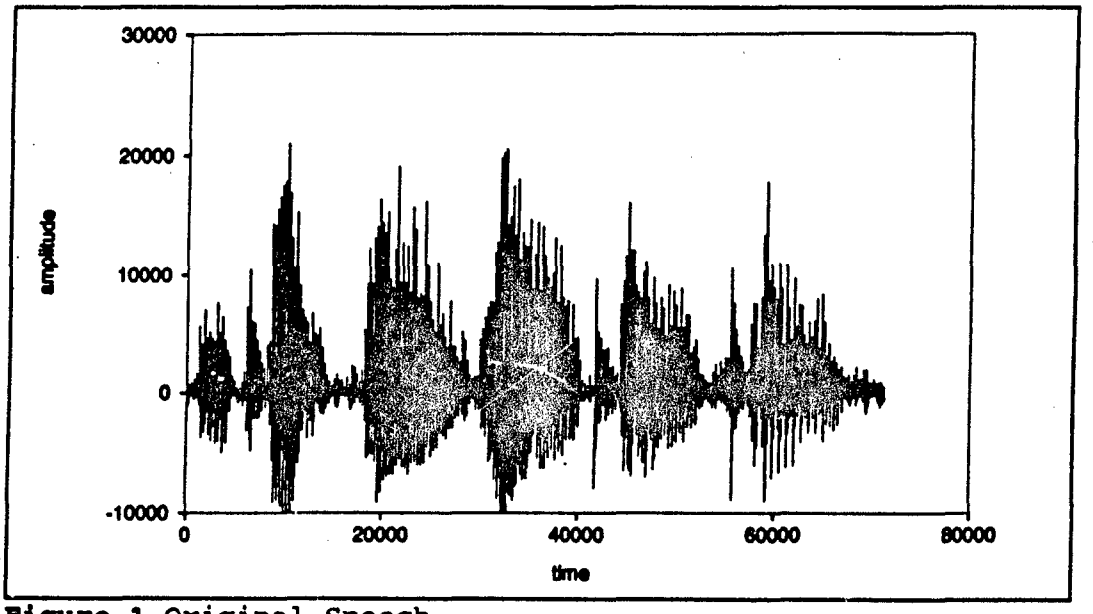

Figure 1 Original Speech

Removing the phase components from the complex frequency vectors produced intelligible speech. That is, the speech could be understood but it was distorted so badly that the speaker could not be recognized. Figures one and two show the original and reproduced speech used for the test.

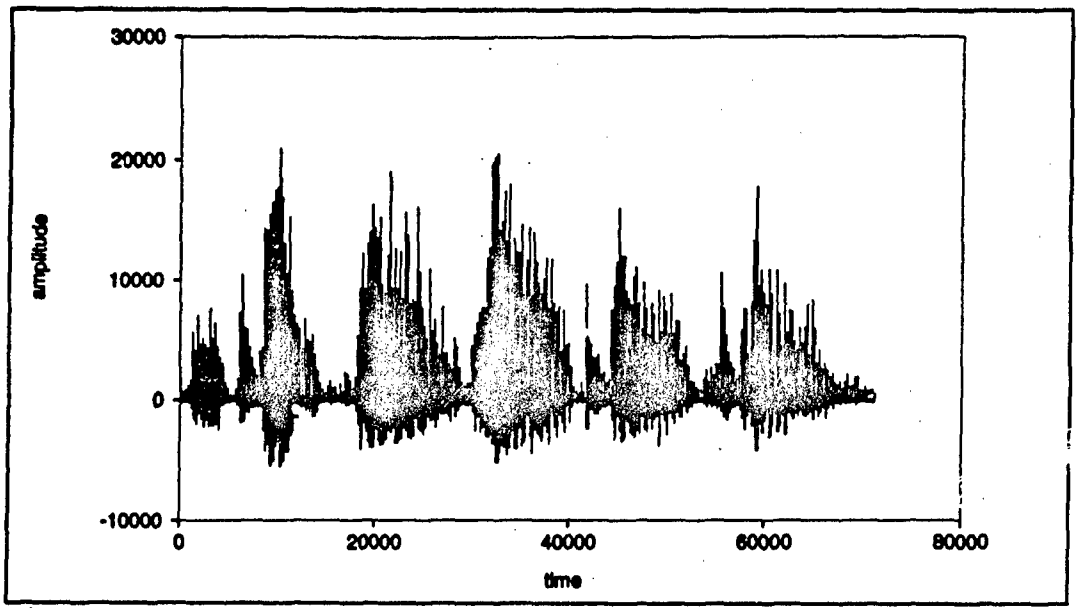

Figure 2 Reproduced Speech, Magnitudes Only

Figures one and two are known to have the same power spectrums. Also, they are similar in that at coincident times there is a similar amount of high and low frequency activity in the reproduced and original waveforms. However, these two signal sound vastly different.

By judging the original and reproduced speech, it is clear that phase does influence quality. Therefore, if a compression technique is to maintain the quality aspects of speech, it should preserve the phase information.

Compression

The compression experiments are similar to those of the previous experiments. Test one compresses the 'magnitude only' spectrum while the second series of compression tests preserves the phase information of the original. Figure three shows the discrete-time plot of the original speech used for the compression tests.

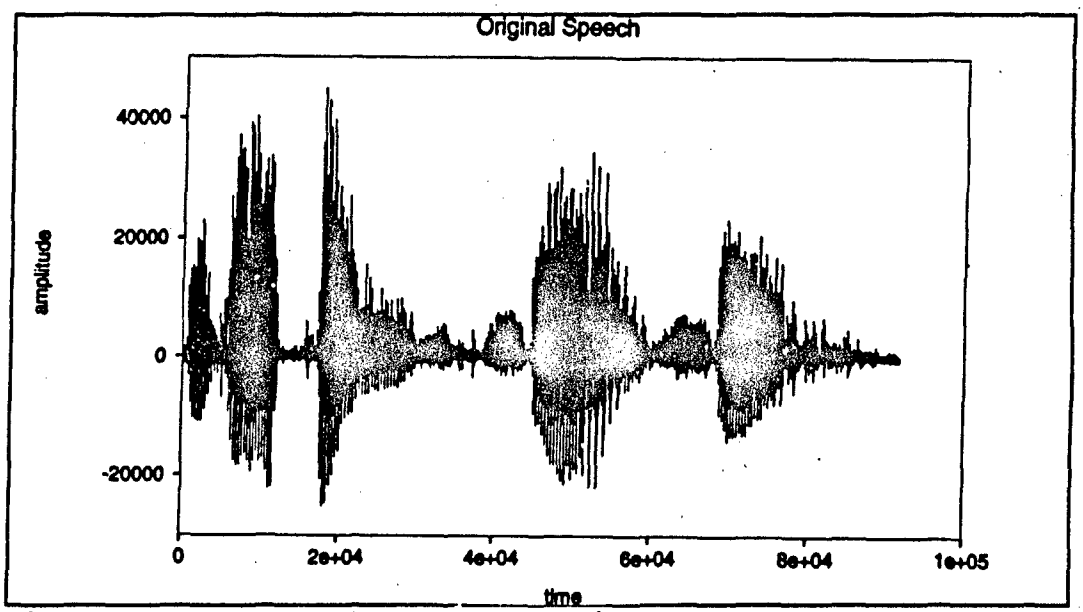

Figure 3 Speech used for Compression Tests

Magnitudes Only

The 'magnitude only' reproductions are shown at figures four and five below. The term "No Compression" means that the KL Transformation did not introduce any compression. However, there is an inherent two to one compression due to the even nature of the 'magnitude only' spectrum. The two to one compression of the KL transform produced the discrete-time waveform of figure five that looks, and sounds, very much like the uncompressed 'magnitude

only' speech of figure four. This result suggests that the KL transformation discards the least important aspects of the speech waveform.

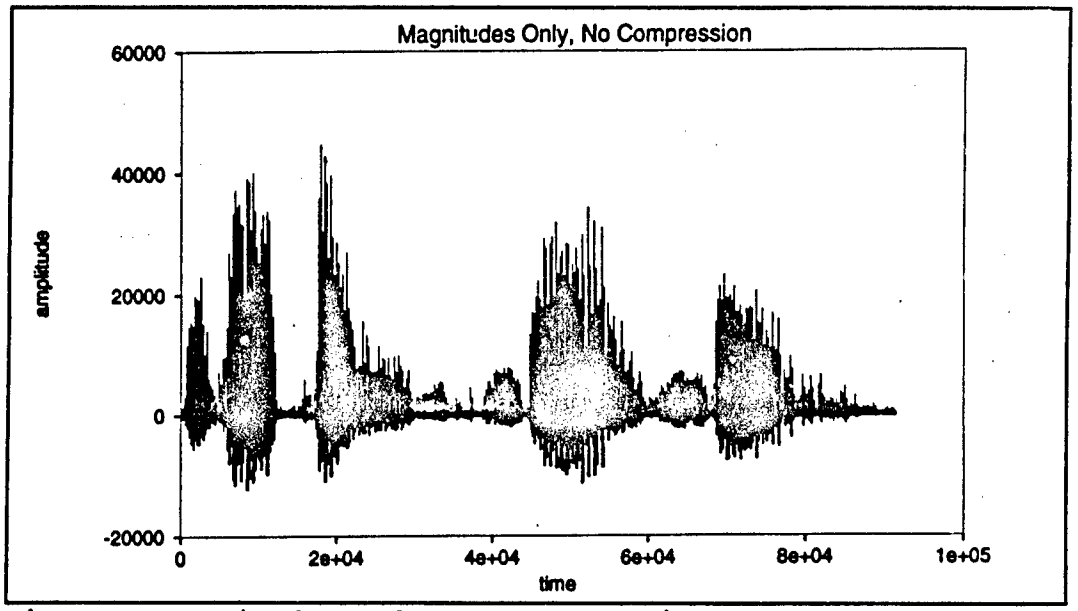

Figure 4 Magnitudes Only, No Compression

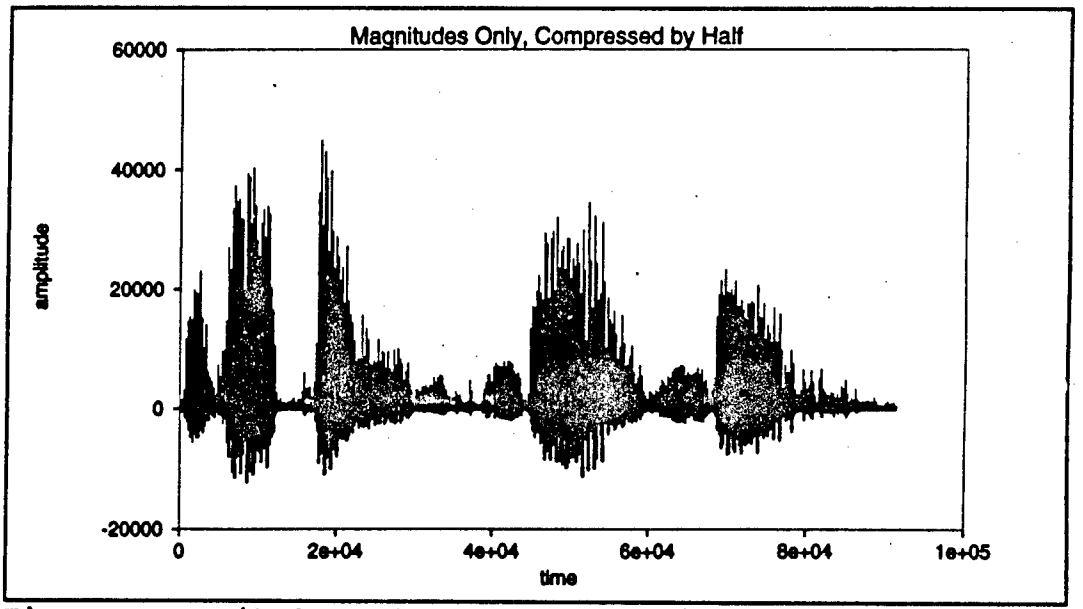

Figure 5 Magnitudes Only, 2:1 Compression

As the 'magnitude only' representation can only at best support intelligible reproductions and the objective is to test phase preservation, no further compression tests of the 'magnitude only' spectrum are performed.

Phase Preserved

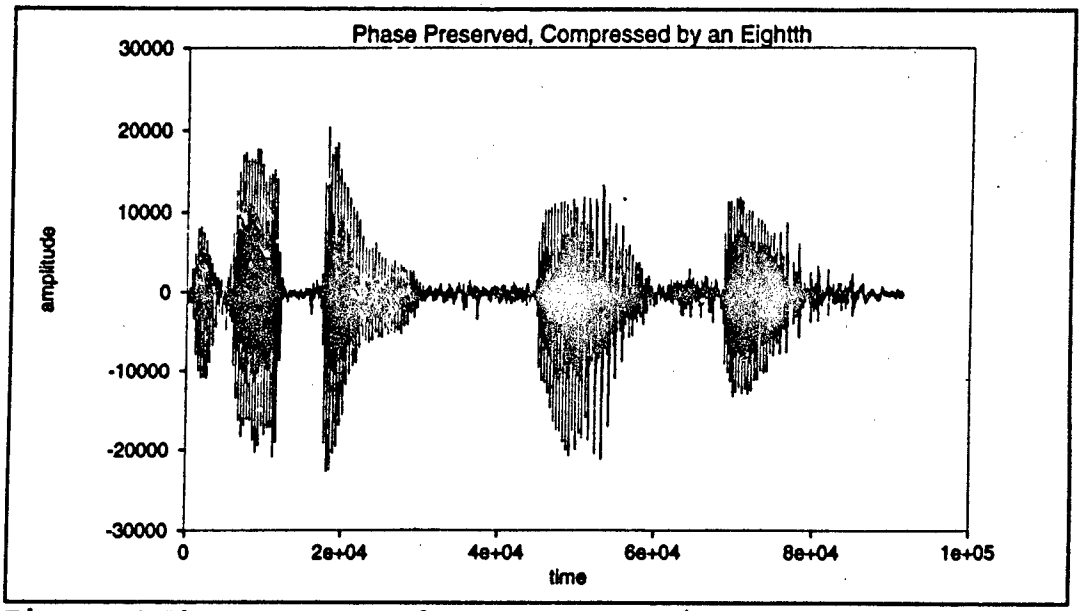

Figure 6 Phase Preserved, 8:1 Compression

Figure six above shows the discrete-time spectrum for an eight to one compression. Notice that the waveform closely resembles the original speech of figure three as there are similar, coincident in time, high and low frequencies. Further, the reproduced waveshape is as balanced about the amplitude equals zero axis as the original speech waveform. The eight to one compression waveform was chosen as it was judged that this compression ratio was the highest for excellent quality reproductions (could not distinguish between original and reproduction). Good quality speech (speaker clearly recognized but speech noisey or distorted) was reproduced for compression ratios up to 20 to one and intelligible speech was achieved for compressions

of approximately 26 to one.

The results of this experiment show that the KL transform will compress speech effectively and that phase can be preserved by representing the complex frequency vector as two vectors of real numbers (one for the imaginary components and one for the real components).

Can the reproduced quality be predicted for a particular compression ratio? Alternatively, can reproduced quality be specified and a corresponding compression ratio selected to meet that specification?

Error Metrics

An error metric needs to be a single number that can be used to achieve repeatable results. The four metrics chosen will each produce two numbers as the complex frequency vectors are represented as a vector pair. These two measurements can be added together to give the single metric.

Mean Square Error

The MSE was measured by averaging the error for each frequency component over all frequency vectors, then taking the magnitude of the mean error vector. The plot of figure seven shows the MSE for an increasing number of coefficients. The reduction in MSE for an increasing number of KL coefficients is consistent with the literature and intuitive as more coefficients increase the degrees of freedom and so 'better' approximations can be supported.

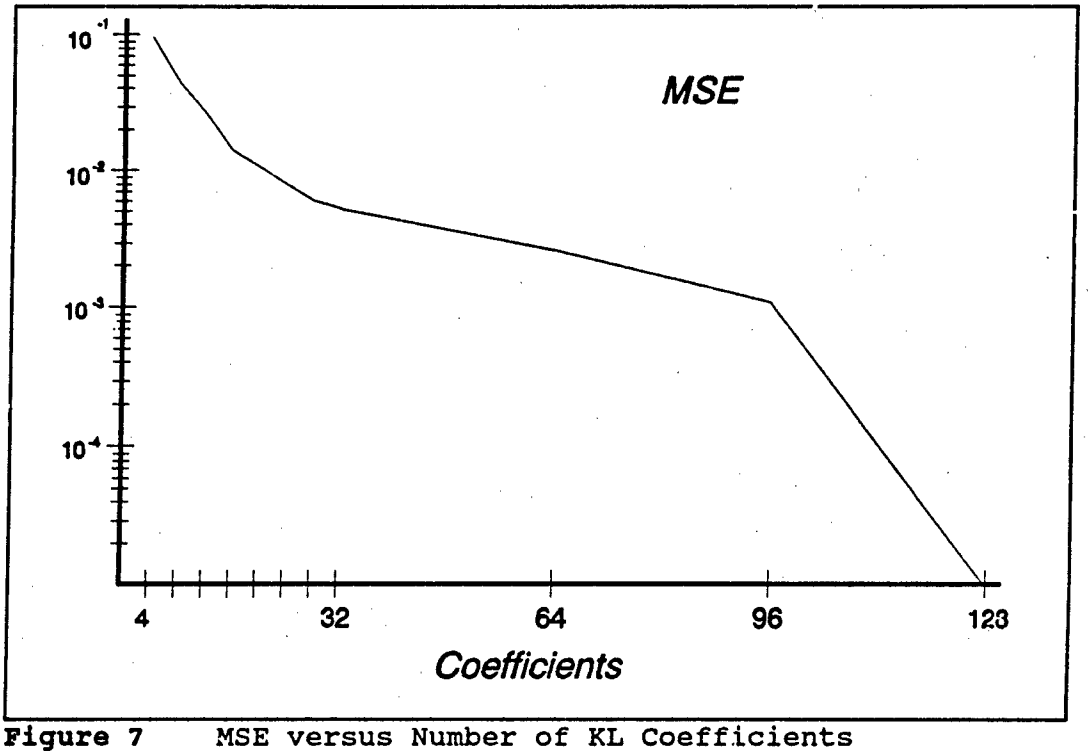

Figure 7 MSE versus Number of KL Coefficients

The problem with the MSE is that the magnitude is only meaningful if the size of the vectors transformed are known.

Relative Mean Square Error

Figure eight shows the Relative MSE for increasing numbers of KL coefficients. Notice that the RMSE increases between 16 and 32 coefficients which is at odds with the previous MSE result. Also, it is known from the previous experiment that the reproduced quality is excellent for that range of KL coefficients. The reason is easily explained by realizing that the RMSE applies an equal weighting to each frequency component of the vector transformed by normalizing each component (see equation seven). Then very low amplitude frequencies, which may contribute little to the quality of the reproductions, can be reproduced with high relative errors but still very low

amplitudes (i.e original = 0.001 and reconstructed = 0.003 produces a relative error of 3.0). Then, clearly, the application of equal weightings to each frequency is not relevant to predicting quality.

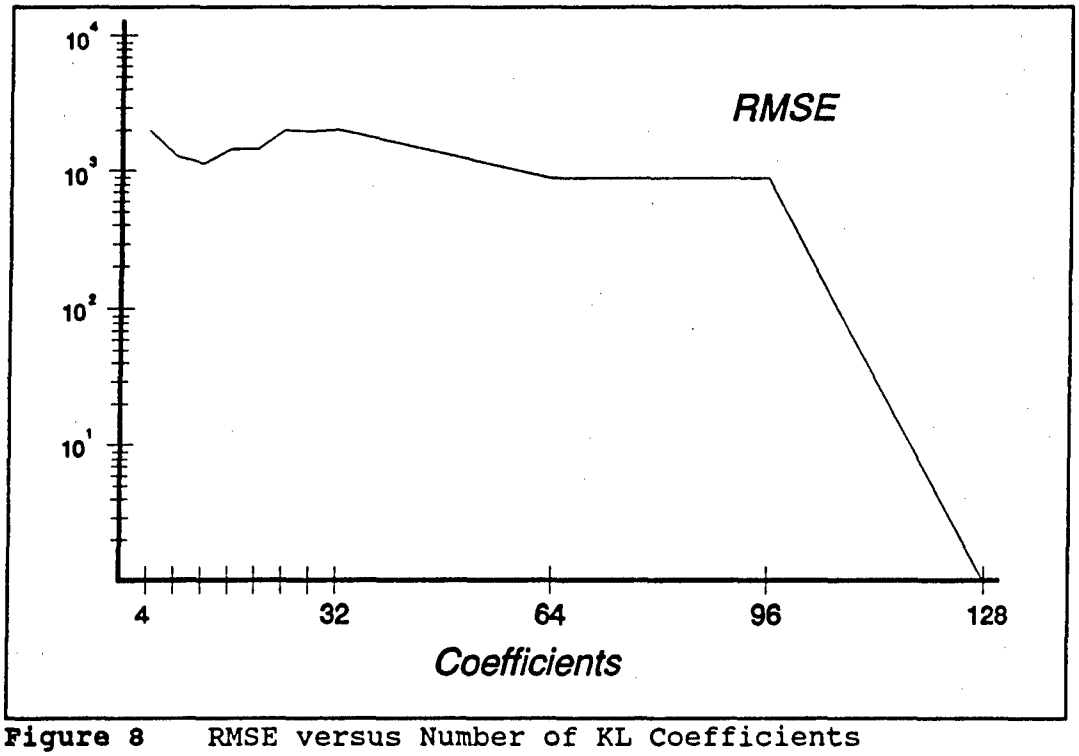

Figure 8 RMSE versus Number of KL Coefficients

The reason for testing the RMSE was to have a metric that represented the relationship between the frequency components of the vectors. That is, specify that all components had to be within, say, 10% and then use a compression ratio that achieved the specified RMSE. However, the plot of figure eight shows that the RMSE can increase, or decrease, for improving quality.

Relative Mean Square Error per Coefficient

After examining the RMSE curve of figure eight, it was decided to

evenly distribute the RMSE over the number of coefficients used for the transformation. This was an attempt at producing a metric that conformed with intuition by decreasing with increasing coefficients. The results obtained were similar to those of the RMSE tests except scaled by the number of coefficients.

Condition Number

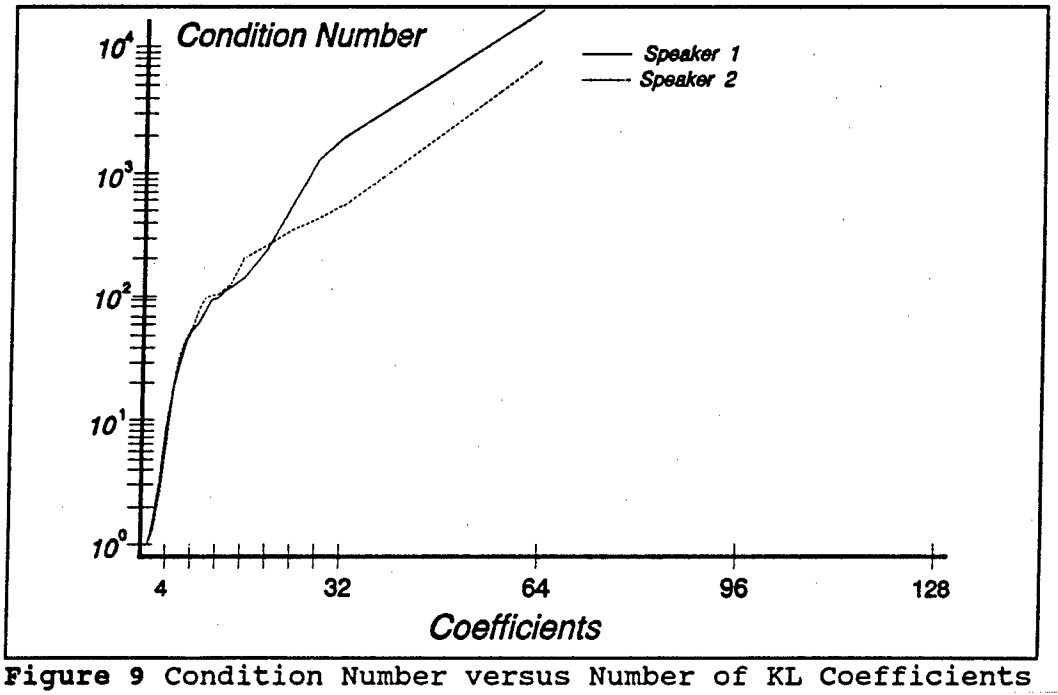

Figure 9 Condition Number versus Number of KL Coefficients

Figure nine shows condition number plotted against the number of KL coefficients. The curve has a slope with a uniform sign (positive slope), so satisfying intuition, and, once the quality levels are associated with a particular CN, compression and quality can be associated. This compression-quality association can be made directly from the decomposed covariance matrix. As an example, consider the threshold for excellent quality established by the phase preservation tests at 16 KL coefficients. Reading

from the curves of figure nine, 16 coefficients corresponds to a CN of 145 for speaker one and 180 for speaker two. If this result held for all speech (if the voice was properly characterized it would), then the number of coefficients ass ciated with a CN of, say, 200 are needed for excellent quality.

Also notice that the CN for the two speakers are close right up to the region where excellent quality begins. There is no perceivable improvement above 16 coefficients, CN of approximately 200, and so the energy associated with the higher order coefficients is low, consequently the CNs vary by greater amounts due to the widely varying low energy frequency components.

Voice Characterization

It was assumed that a speaker can be characterized by a comprehensive set of phonemes. To test the concept the system will be trained on one set of words and then used reproduce a second set, made up of similar utterances but different words. If the second set are reproduced as accurately as the training set, the assumption will be considered reasonable. However, a much more exhaustive study using a comprehensive training set will still be required. As a matter of interest the technique will also be used on multiple speakers.

Single Speaker

An individual KL transform was computed for each of the two speakers and used to transform the testing sentence of Appendix H. The

results of each test were the same. That is, the threshold for excellent quality occurred at 16 coefficients (note the similarity with the phase preservation experiments).

One point of interest was the reproduction of the fricative /f/ in the word fly. As the number of coefficients were decreased below 16, the quality of the reproduction became noisier as in the phase preservation experiment. However, for this experiment, the /f/ sound became noisier for more coefficients than the previous tests. Apart from that, the testing sentence was transformed as well as any of the sentences from the training set.

Multiple Speakers

Figure nine shows the energy curve for two speakers which was derived from each one speaking the eight training sentences of Appendix H. When four or more coefficients are used, the energy levels are within 1% of each other. When more than eight are used the two energy curves are less than 0.1% from each other.

When the dual speaker KL Transform was used on each speaker's testing sentence (Appendix H), the resulting reproduced quality sounded no different from that achieved when each of the single speaker KL Transforms were used. That is, the threshold of excellent quality was again 16 coefficients and the fricative /f/ was distorted more than the voiced sounds when less than 16 coefficients were used.

Summary

Removing the phase information from frequency vectors and only using the magnitudes will produce intelligible speech but not toll quality speech.

Excellent quality can be achieved with compression ratios of up to eight to one by preserving phase and using the KL transform.

The MSE decreases with decreases in compression ratio; however, the RMSE and RMSE per Coefficient do not vary consistently with compression ratio.

The condition number is related to the signal energy ranked by the KL Transform and increases with decreasing compression ratio.

Words can be transformed without the compression model being specifically trained on those words. That is, the training set need only contain the sub-words (utterances, syllables, etc) to be transformed.

The KL Transform compression model could also transform multiple voices.

CHAPTER FIVE

CONCLUSIONS AND RECOMMENDATIONS

Influence of Phase on Quality

A complex Fourier transform produces, for a set of frequencies, magnitudes which represent the amount of energy at a particular frequency and phase which represents the time relationship between the frequencies of the set. Consequently, when the magnitudes are maintained and the phase set to zero, the time relationships between the frequency components of the utterance is lost. When this time relationship is discarded, toll quality speech (i.e. no processing) is reduced to intelligible speech. Therefore, phase does influence the reproduced quality of speech.

There are at least two implications of this conclusion. Firstly, if toll quality communications is to be achieved with compressed speech, some method for preserving the phase information is needed. The second implication concerns speech recognizors. If speakers are to be recognized then preserving the phase information will be required. For speaker dependent recognizors, including the phase information in the training pattern (or equivalent technique) should enhance performance as there are more degrees of freedom with which to separate individual patterns.

Further to the phase preservation results is the technique used to preserve phase. Representing a vector of complex numbers as a vector pair of real numbers proved effective. As each vector of the pair is reproduced with minimum MSE, then reconstructing with this pair provides a close representation of the original speech. The utility of the technique is it is not as computationally expensive as working with a KL Transform consisting of complex eigenvectors for rows and a frequency vector consisting of complex

components. However, phase preservation may not be suitable for communications applications as it might be difficult to transmit the 33 coefficients over narrowband channels.

Error Metrics

Of the four error metrics investigated only the MSE and ER were sensible proposals. The MSE metric is not suitable as it requires some prior knowledge of the amplitude of the original speech for the MSE measurement to be useful. This prior knowledge could be obtained after the compression system is trained. Testing data (different to that of the training set) could be passed through the system; the average MSE measured for the range of compression ratios; and, then a compression ratio that matches the required quality performance selected.

The CN is probably the most useful metric. Assuming the speaker's voice is properly characterized, then the CN can be derived directly form the training set without any testing data being required. Although, some prior knowledge of the association between quality levels and CN is necessary. A suitable compression ratio can then be selected according to the performance criteria. The results of the CN experiments agree with those reported by Chen and Huo[16] (i.e. 98% energy needed for excellent quality) as a CN of 200 corresponds to approximately 99% of the speech energy.

Voice Characterization

Voices can be characterized by training sets that consist of sub-word utterances. This means that a speaker dependent compression model can be developed on a training set of sub-words and, so long as those sub-words are representative of the speech to be compressed, subsequent speech can be compressed and reproduced with toll quality.

The implication of the characterization result is that speech can be preprocessed using the KL Transformation into a space of smaller dimension and not lose any of its quality. Consequently, applications such as speech recognition and speaker identification can be performed in the reduced space.

The examples produced by this thesis show that speech vectors of 256 discrete-time samples can be represented with 33 coefficients. The number of coefficients can be predicted by the CN metric and experience. That is, toll quality requires a CN of 200 which is represented by 16 coefficients in each vector of the pair plus the DC value.

Optimum Speech Transform

The KL Transform performed as predicted by the literature in that Chen and Huo's[16] results were reproduced and MSE decreased with increasing KL coefficients as predicted Tou and Gonzalez[15]. So it is clear that the KL Transform is a suitable technique for speech compression.

A result of interest is that the number of KL coefficients was of

only 1/8th the number of the Fourier coefficients representing the original speech vector. Therefore, the Fourier Transform is not the optimal means for representing the original speech.

The optimal means for representing the original speech, might be found from the KL Transformation. Assume that a voice has been completely characterized by some training set of frequency vectors (complex or pairs of reals). Singular Value Decomposition (or some other eigen-decomposition) produces the directions of maximum variance (eigenvectors) through the space (or spaces) and the energy associated with each direction (singular values) is also known. In order to generate the covariance matrix the average spectrum is determined. The question becomes - Is there enough information in the directions of maximum variance and the average spectrum with which to determine the optimum transform?

The answer is beyond the scope of this thesis but it might be that the optimal transform can be found analytically. A completely invertible compact transform where the energy was spread evenly among the coefficients would be the desired result.

Recommendations

The recommendations concern applications involving the KL Transform and areas for further study.

Applications

In order to achieve quality reproductions, the phase information

associated with the complex frequency vectors needs to preserved.

For applications such as speaker dependent speech recognition and speaker identification, the KL transform can be used to achieve feature reduction (i.e. KL Transform the Fourier Transform) and without loss of quality.

Communications applications should only use the KL Transform techniques when the application allows speaker independence to be traded off against reproduced quality.

The storage of large quantities of speech data could also use the technique as parts of the speech could be used to characterize the voice and the whole speech sequence transformed for storage.

Further Study

There are two areas for further study. Firstly, the phonemic characterization of voices should be investigated further as the potential exists for training sets to be developed independent of the application.

Secondly, finding the optimal transform for speech would allow more efficient compression models to be developed.

BIBLIOGRAPHY

1. Lathi, B. P. *Modern Digital and Analog Communications Systems,* New York: Holt-Sanders, 1983.

2. Rabiner, Lawrence R and Gold, Bernard. *Theory and Application of Digital Signal Processing,* Englewood-Cliffs: Prentice-Hall Incorporated, 1975.

3. Leen Todd, Rudnick Mike and Hammerstrom Dan. "Hebbian Feature Discovery Improves Classifier Efficiency," *International Joint Conference on Neural Networks,* Vol 1:51, 1990

4. Weinstein, Clifford J. "Opportunities for Advanced Speech Processing in Military Computer-Based Systems," *Proceedings of the IEEE, 79: 1626-1641 (November 1991).*

5. Parsons, Thomas W. *Voice and Speech Processing.* New York: McGraw-Hill Book Company, 1987.

6. Strobach, Peter. *Linear Prediction Theory.* Berlin: Springer-Verlag, 1990

7. Flanagan, J. L. "Voices of Men and Machines," *Journal of the Acoustic Society of America,* 51: 1375-1387, March 1972.

8. Makhoul, John. "Linear prediction: A Tutorial Review." *Proceedings of the IEEE, 63: 561-580* April 1975.

9. White, George M. "Speech Recognition: A tutorial Overview," *IEEE Computer,* 9:40-53, May 1976.

10. Gaskill, Jack D. *Linear systems, Fourier Transforms, and Optics.* New York: John Wiley and Sons, 1978.

11. Stremler, Ferrel G. *Introduction to Communication Systems,* Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

12. Yuen, C. K. and Fraser, D. *Digital Spectral Analysis.* Melbourne: CSIRO and London: Pitman Publishing Company, 1979.

13. Switzer, Shane. *Frequency Domain Speech Coding.* MS Thesis, AFIT/GE/ENG/91D-10. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991 (AAI-7101).

14. Pols, Louis C. W. "Real-Time Recognition of Spoken Words," *IEEE Transactions on Computing C-20: 972-978, September 1971.*

15. Tou, Julius T. and Gonzalez, Rafael C. *Pattern Recognition Principles.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1974.

16. Chen, C. S. and Huo, K. -S. "Karhunen-Loeve method for data compression and speech analysis," IEE Proceedings-I 138: 377-380 October 1991.

17. Rabiner, L. I. and Schafer, R. W. *Digital Processing od Speech Signals,* Englewood Cliffs: Prentice-Hall, 1978.

18. Vetterli, Martin. "Wavelets and Filter Banks: Theory and Design," *IEEE Transactions on Signal Processing,* 40: 2207-2232, September 1992.

19. Kabrisky, M. "Lectures in Speech Recognition," AFIT, April 1992.

20. Rogers, Steven K. and Kabrisky, Matthew. *An Introduction to Biological and Artificial Neural Networks for Pattern Recognition.* Bellingham: SPIE Optical Engineering Press, 1991.

21. Brown, Michael K. and Wilpon, Jay G. "A Grammar Compiler for Connected Speech Recognition," *IEEE Transactions on Signal Processing,* 39: 17-28, January 1991.

22. Press, William H, Flannery, Brian P, Teukolsky, Saul A and Vetterling, William T. *Numerical Recipes in C, The Art of Scientific Computing,* Cambridge (UK): Cambridge University Press, 1988.

23. Horn, Roger A. and Johnson, Charles R. *Matrix Analysis,* Cambridge(UK): Cambridge University Press, 1985.

APPENDIX A

```
/*              PHASE REMOVAL SIMULATION: WITHOUT COMPRESSION

          Command Line Inputs : remove speech.dat

     The program reads the data file of discrete-time samples and fourier
transforms the data to complex frequency vectors.  The magnitudes of the complex
components are determined and loaded back into the real locations of the FFT
buffer while the imaginary locations are set to zero.

     The inverse fourier transform is performed and the reconstructed data
written out to a file.

          Outputs : nophase.dat

     Program written by FLTLT Don Dryley at AFIT September 1992.
*/


#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define MAX_AMP 32768
#define N 256
#define n 128



void main(argc,argv)
int argc;
char *argv[];
{
     /* DECLARE VARIABLES */
     FILE *inhandle,*outhandle;
     int i, j, k, vectors, integers_read, tempint;
     long *buff;
     double *d, *data_mag, DC;
     char  outfile[] = "nophase.dat";
```

```
/* CHECK ARGUMENTS */

if( argc != 2)
{
      printf("Format: D>KLT source.xxx");
      exit(-1);
}



/* CREATE ARRAYS, INITIALISED TO ZERO */

buff = lvector(1,N);        /* input buffer */
d = dvector(1,2*N);         /* FFT of one frame of speech is complex */
data_mag = dvector(1,n);    /* Single frequency vector of magnitudes */



/* NEED TO KNOW HOW MANY VECTORS ARE IN THIS FILE */

/* open input file */
inhandle = fopen(argv[1],"r");
if(inhandle == NULL)
{
      printf("Can't open file %s." ,argv[1]);
      exit(-1);
}

vectors = integers_read = 0;          /* input counter */

/* count vectors */
while(fscanf(inhandle,"%d",&tempint) != EOF)
{
          integers_read++;

      /* increment input counter if whole vector was read */
      if (integers_read == N)
    {
            vectors++;
        integers_read = 0;
    }
}

rewind(inhandle); /* reset input file */
```

```
/* PERFORM EXPERIMENT */

/* open output file */
outhandle = fopen(outfile,"w");
if(outhandle == NULL)
{
      printf("Can't open file %s." ,outfile);
      exit(-1);
}

integers_read = 0;  /* set integer counter */

/* re - read whole file */
k = 1;
while(k <= vectors)
{
      fscanf(inhandle,"%d",&tempint);
      integers_read++;
      buff[integers_read] = tempint;

      if (integers_read == N)
      {
            /* convert input characters to floats */
            for (i=1; i<=integers_read; i++)
               {
                  /* scale data for an array of floats between 0 and 1*/
                  d[2*i-1] = (double) buff[i];
                      d[2*i-1] /= MAX_AMP;
               }

            /* compute complex fft */
            four1(d,N,1); /* overwrite input array */

            /* convert rectangular to polar and keep magnitudes */
            mag(d,n,data_mag,DC);

            /* clear fft buffer */
            for (i=1; i<=2*N; i++)
                  d[i]=0.0;

            /* load real locations of fft array */
            for (i=0; i<=n; i++)
                  if (i==0)
                        d[i+1] = DC;
                  else d[2*i+1] = data_mag[i];
            for (i=1; i<n; i++)
                  d[n+2*i+1] = data_mag[n-i];

            /* compute inverse fft */
            four1(d,N,-1);
```

```
                    /* write reconstructed data to to output file */
                    for (i=1; i<=N; i++)
                       {
                            d[2*i-1] *= MAX_AMP;
                            d[2*i-1] /= N;
                        fprintf(outhandle,"%d\n",(int) d[2*i-1]);
                       }

                    /* clear fft buffer */
                    for (i=1; i<=2*N; i++)
                         d[i]=0.0;

                    integers_read = 0;

                    k++;
               }
          }
          fclose(inhandle);
          fclose(outhandle);
}
```

APPENDIX B

```
/*                PHASE REMOVAL SIMULATION: WITH COMPRESSION

            Command Line INPUTS :  compmag source file

      Program reads frames of speech from the source file and fourier transforms
 the N time samples into N/2 frequency magnitudes.  These magnitudes form vectors
 from which a covariance matrix is computed.

      The covariance matrix is decomposed, using Singular Value Decomposition
 (SVD), into a vector of SVs and two orthogonal matrices of eigenvectors.  The
 eigenvectors are ranked according to the magnitude of the associated SV
 (squareroot of eigenvalue) and used as the rows of the KL transform matrix.

      The inverse KL transform matrix is the transpose of the KL transform.


            OUTPUTS : File KLT.dat
                    : File averages.dat


      Program written by FLTLT Don Dryley at AFIT during August 1992.
*/


#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define MAX_AMP 32768
#define N 256
#define n 128



void main(argc,argv)
int argc;
char *argv[];
{
      /* DECLARE VARIABLES */
      FILE *inhandle,*outhandle;
      int i, j, k, vectors, integers_read, tempint, dim;
      long *buff;
      double *d, *data_mag, *average, **A, *W, **V, DC, *KL;
      char outfile[] = "witcomp.dat";
```

```
/* PROMPT FOR NUMBER OF KL COEFFICIENTS */

printf("Enter number of coefficients <1 - %d> ... ",n);
scanf("%d",&dim);
printf("\n\n");



/* CHECK ARGUMENTS */

if( argc != 2)
{
      printf("Format: D>KLT source.xxx");
      exit(-1);
}



/* CREATE ARRAYS, INITIALISED TO ZERO */

buff = lvector(1,N);        /* input buffer */
d = dvector(1,2*N);         /* FFT of one frame of speech is complex */
data_mag = dvector(1,n);    /* Single frequency vector of magnitudes */
average = dvector(1,n);     /* average of discrete frequencies */
A = dmatrix(1,n,1,n);       /* Covariance Matrix gets over written */
V = dmatrix(1,n,1,n);       /* Matrix for right singular vectors */
W = dvector(1,n);           /* Matrix of singular values */
KL = dvector(1,dim);        /* matrix of KL coefficients */



/* NEED TO KNOW HOW MANY VECTORS ARE IN THIS FILE */

/* open input file */
inhandle = fopen(argv[1],"r");
if(inhandle == NULL)
{
      printf("Can't open file %s." ,argv[1]);
      exit(-1);
}

vectors = 0;          /* input counter */
integers_read = 0;    /* integer counter */

/* count vectors and find average of each vector component */
while(fscanf(inhandle,"%d",&tempint) != EOF)
{
      integers_read++;
      buff[integers_read] = tempint;
```

```
        /* increment input counter if whole vector was read */
        if (integers_read == N)
        {
              vectors++;        /* update new vector */

              /* convert input characters to floats */
              for (i=1; i<=integers_read; i++)
                 {
                    /* scale data for an array of floats between 0 and 1*/
                    d[2*i-1] = (double) buff[i];
                       d[2*i-1] /= MAX_AMP;
                 }

              /* compute complex fft */
              four1(d,N,1); /* overwrite input array */

              /* convert rectangular to polar and keep magnitudes */
              mag(d,n,data_mag,DC);

              /* accumulate like components */
              for (i=1; i<=n; i++)
                    average[i] += data_mag[i];

              /* clear input array */
              for (i=1; i<=2*N; i++)
                    d[i]=0.0;

              integers_read = 0; /* reset for next vector */
        }
) /* file flushed */


/* COMPUTE MEANS */

for (i=1; i<=n; i++)
      average[i]=average[i]/vectors;



/* DETERMINE COVARIANCE MATRIX */

/* ensure input array cleared */
for (i=1; i<=2*N; i++)
      d[i]=0;

/* reset input file */
rewind(inhandle); /* close input file */

integers_read = 0;
```

```
/* re - read whole file */
k = 1;
while(k <= vectors)
{
      fscanf(inhandle,"%d",&tempint);
      integers_read++;
      buff[integers_read] = tempint;

      if (integers_read == N)
      {
            /* convert input characters to floats */
            for (i=1; i<=integers_read; i++)
               {
                  /* scale data for an array of floats between 0 and 1*/
                  d[2*i-1] = (double) buff[i];
                      d[2*i-1] /= MAX_AMP;
               }

            /* compute complex fft */
            four1(d,N,1); /* overwrite input array */

            /* convert rectangular to polar and keep magnitudes */
            mag(d,n,data_mag,DC);

            /* subtract mean from each component */
            for (i=1; i<=n; i++)
                  data_mag[i] = data_mag[i] - average[i];

            /* update covariance matrix */
            for (i=1; i<=n; i++)
                  for (j=1; j<=n; j++)
                        A[i][j] += data_mag[i] * data_mag[j];

            /* clear input array */
            for (i=1; i<=2*N; i++)
                  d[i] = 0;

            integers_read = 0;

            k++;
      }
}


/* DECOMPOSE COVARIANCE MATRIX */

svdcmp(A,n,n,W,V);
eigsrt(W,A,n); /* columns of A are ranked rows of KL transform matrix */
```

```
/* COMPRESS SPEECH USING KL TRANSFORM */

/* open output file */
outhandle = fopen(outfile,"w");
if(inhandle == NULL)
{
      printf("Can't open file %s." ,outfile);
      exit(-1);
}

/* ensure input array cleared */
for (i=1; i<=2*N; i++)
      d[i]=0;

/* reset input file */
rewind(inhandle); /* close input file */

integers_read = 0;

/* re - read whole file */
k = 1;
while(k <= vectors)
{
      fscanf(inhandle,"%d",&tempint);
      integers_read++;
      buff[integers_read] = tempint;

      if (integers_read == N)
      {
            /* convert input characters to floats */
            for (i=1; i<=integers_read; i++)
               {
                  /* scale data for an array of floats between 0 and 1*/
                  d[2*i-1] = (double) buff[i];
                      d[2*i-1] /= MAX_AMP;
               }

            /* compute complex fft */
            four1(d,N,1); /* overwrite input array */

            /* convert rectangular to polar and keep magnitudes */
            mag(d,n,data_mag,DC);

            /* clear fft buffer */
            for (i=1; i<=2*N; i++)
                  d[i]=0.0;

            /* subtract mean from each component */
            for (i=1; i<=n; i++)
                  data_mag[i] -= average[i];
```

```
                    /* compress data */
                    for (i=1; i<=dim; i++)
                    {
                          KL[i] = 0.0;
                          for (j=1; j<=n; j++)
                                KL[i] += A[j][i] * data_mag[j];
                    }

                    /* expand data by inverse transformation */
                    for (i=1; i<=n; i++)
                    {
                          data_mag[i] = 0.0;
                          for (j=1; j<=dim; j++)
                                data_mag[i] += A[i][j] * KL[j];
                    }

                    /* add mean of each component */
                    for (i=1; i<=n; i++)
                          data_mag[i] += average[i];

                    /* load real locations of fft array */
                    for (i=0; i<=n; i++)
                          if (i==0)
                                d[i+1] = DC;
                          else d[2*i+1] = data_mag[i];
                    for (i=1; i<n; i++)
                          d[n+2*i+1] = data_mag[n-i];

                    /* compute inverse fft */
                    four1(d,N,-1);

                    /* write reconstructed data to to output file */
                    for (i=1; i<=N; i++)
                       {
                             d[2*i-1] *= MAX_AMP;
                             d[2*i-1] /= N;
                          fprintf(outhandle,"%d\n",(int) d[2*i-1]);
                       }

                    /* clear input array */
                    for (i=1; i<=2*N; i++)
                          d[i] = 0;

                    integers_read = 0;

                    k++;
              }
        }

        fclose(inhandle);
        fclose(outhandle);
}
```

APPENDIX C

LISTING ONE

```
/*                  AVERAGES MATRICES CONSTRUCTION PROGRAM

            Command Line INPUTS :  average speechfile.dat

      Program tests if the two average files exist.  If the files do not exist
they are created and two 128 element column vectors are loaded with zeros.

      If (when) the files exist the speech file, specified at the command line,
is opened and discrete-time samples are fourier transformed to generate vectors
of N complex fourier coefficients.

      The complex vectors are separated into two real vectors, one for the real
components and one for the imaginary components.  These vectors are used to
update the two average vectors.


            Outputs provided : avgs_Re.dat
                             : avgs_Im.dat


      Written by FLTLT Don Dryley at AFIT Sep 1992
*/


#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define N 256
#define n 128


void main(argc,argv)
int argc;
char *argv[];
{
      FILE        *inhandle,*out1handle,*out2handle;
      int         i, j, k, x, MAX_AMP, vectors, New_vectors, integers_read,
                  tempint;
      long        *buff;
      float       tempfloat;
      double      *d, *data_R, *data_I, *averageR, *averageI, DC, zero = 0.0;
      char        averagesR[] = "avgs_Re.dat",
```

```
            averagesI[] = "avgs_Im.dat";

/* CHECK ARGUMENTS */

if( argc != 2)
{     printf("Format: D>Cov_C source.xxx");
      exit(-1);
}



/* CREATE ARRAYS, INITIALISED TO ZERO */

buff = lvector(1,N);          /* input buffer */
d = dvector(1,2*N);           /* FFT of one frame of speech is complex */
data_R = dvector(1,n);        /* Single vector of real coefficients */
data_I = dvector(1,n);       /* Single vector of imaginary coefficients */
averageR = dvector(1,n);      /* average of real coefficients */
averageI = dvector(1,n);      /* average of imaginary coefficients */



/* OPEN AVERAGES FILES */

/* open file of Real averages */
if ((outlhandle = fopen(averagesR,"r")) != NULL)
{
      /* file exists */
      fclose(outlhandle);
      outlhandle = fopen(averagesR,"r+");
      if (outlhandle == NULL)
      {
            printf("Can't open file %s, Exiting to system\n",averagesR);
            exit(-1);
      }
}
else
{
      /* create file for writing and reading */
      outlhandle = fopen(averagesR,"w+");
      if (outlhandle == NULL)
      {
            printf("Can't open file %s, Exiting to system\n",averagesR);
            exit(-1);
      }

      /* write zero averages and zero vectors to file */
      for (i=1; i<=n; i++)
            fprintf(outlhandle,"%e\n",(float)zero);
      fprintf(outlhandle,"%d\n",(int) zero);

      /* reset pointer to start of file */
      rewind(outlhandle);
```

```
}


/* LOAD REAL ARRAY FROM FILE */

for (i=1; i<=n; i++)
{
      fscanf(out1handle,"%e",&tempfloat);
      averageR[i] = (double) tempfloat;
}

/* read number of contributing vectors */
fscanf(out1handle,"%d",&vectors);

/* reset pointer to start of file */
rewind(out1handle);

/* scale averages back to accumulated sums */
for (i=1; i<=n; i++)
      averageR[i] *= vectors;

/* open file of Imaginary averages */
if ((out2handle = fopen(averagesI,"r")) != NULL)
{
      /* file exists */
      fclose(out2handle);
      out2handle = fopen(averagesI,"r+");
      if (out2handle == NULL)
      {
            printf("Can't open file %s, Exiting to system\n",averagesI);
            exit(-1);
      }
}
else
{
      /* create file for writing and reading */
      out2handle = fopen(averagesI,"w+");
      if (out2handle == NULL)
      {
            printf("Can't open file %s, Exiting to system\n",averagesI);
            exit(-1);
      }

      /* write zero averages and zero vectors to file */
      for (i=1; i<=n; i++)
            fprintf(out2handle,"%e\n",(float)zero);
      fprintf(out2handle,"%d\n",(int) zero);

      /* reset pointer to start of file */
      rewind(out2handle);
}
```

```
/* LOAD IMAGINARY ARRAY FROM FILE */

for (i=1; i<=n; i++)
{
      fscanf(out2handle,"%e",&tempfloat);
      averageI[i] = (double) tempfloat;
}

/* reset pointer to start of file */
rewind(out2handle);

/* scale averages back to accumulated sums */
for (i=1; i<=n; i++)
      averageI[i] *= vectors;



/* OPEN SPEECH DATA FILE */

inhandle = fopen(argv[1],"r");
if(inhandle == NULL)
{
      printf("Can't open file %s." ,argv[1]);
      exit(-1);
}

/* read entire file and find maximum amplitude */
fscanf(inhandle,"%d",&MAX_AMP);
while(fscanf(inhandle,"%d",&tempint) != EOF)
{
      if (abs(MAX_AMP) < abs(tempint))
            MAX_AMP = tempint;
}
rewind(inhandle);



/* FOURIER TRANSFORM SPEECH DATA */

for (x=1; x<=2; x++)
{
      if (x==2) /* offset pointer for 50% overlap */
            for (i=1; i<=n; i++)
                  fscanf(inhandle,"%d",&tempint);

      /* count vectors and find average of each vector component */
      integers_read = New_vectors = 0;
      while(fscanf(inhandle,"%d",&tempint) != EOF)
      {
            integers_read++;
            buff[integers_read] = tempint;
```

```
                /* increment input counter if whole vector was read */
                if (integers_read == N)
                {
                     New_vectors++;        /* update new vector */
                     vectors++;

                     /* convert input characters to doubles */
                     for (i=1; i<=N; i++)
                     {
                          d[2*i-1] = (double) buff[i];
                          d[2*i-1] /= abs(MAX_AMP);
                     }

                     /* compute complex fft */
                     fourl(d,N,1); /* overwrite input array */

                     /* collect real and imaginary vectors, ignore DC */
                     for (i=1; i<=n; i++)
                     {
                          data_R[i] = d[2*i+1];
                          data_I[i] = d[2*i+2];
                     }

                     /* accumulate like components */
                     for (i=1; i<=n; i++)
                     {
                          averageR[i] += data_R[i];
                          averageI[i] += data_I[i];
                     }

                     /* clear input array */
                     for (i=1; i<=2*N; i++)
                          d[i]=0.0;

                     integers_read = 0; /* reset for next vector */
                }

          }
          rewind(inhandle);
}


/* UPDATE AVERAGES AND SAVE TO FILES */

for (i=1; i<=n; i++)
{
     averageR[i]/=vectors;
     averageI[i]/=vectors;
}
for (i=1; i<=n; i++)
     fprintf(outlhandle,"%e\n",averageR[i]);
fprintf(outlhandle,"%d\n",vectors);
```

```
        fclose(out1handle);
        for (i=1; i<=n; i++)
              fprintf(out2handle,"%e\n",averageI[i]);
        fprintf(out2handle,"%d\n",vectors);
        fclose(out2handle);
}
```

LISTING TWO

```
/*          COVARIANCE AND AVERAGES MATRICES CONSTRUCTION PROGRAM

            Command Line INPUTS :  cov speechfile.dat

      Program tests if the covariance files exist.   If the files do not exist
they are created and loaded with zeros in a 512 x 512 matrix.   If the files
exist the speech file is opened and used to update the covariance matrix.

      Associated with the covariance matrices are files of averages which holds
the average value of the complex frequency components used to build the
covariance matrices.


            Outputs provided : covar_Re.dat
                             : covar_Im.dat



      Written by FLTLT Don Dryley at AFIT Sep 1992
*/


#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define N 256
#define n 128


void main(argc,argv)
int argc;
char *argv[];
{
      FILE        *inhandle,*outlhandle,*out2handle;
      int         i, j, k, x, MAX_AMP, vectors, New_vectors, integers_read,
                  tempint;
      long        *buff;
      float       tempfloat;
      double      *d, *data_R, *data_I, *averageR, **A_R, *averageI, **A_I, *W,
                  **V, **Cov, DC, zero = 0.0;
      char        averagesR[] = "avgs_Re.dat",
                  averagesI[] = "avgs_Im.dat",
                  covarianceR[] = "covar_Re.dat",
```

```
            covarianceI[] = "covar_Im.dat";

/* CHECK ARGUMENTS */

if( argc != 2)
{     printf("Format: D>Cov_C source.xxx");
      exit(-1);
}



/* CREATE ARRAYS, INITIALISED TO ZERO */

buff = lvector(1,N);       /* input buffer */
d = dvector(1,2*N);        /* FFT of one frame of speech is complex */
data_R = dvector(1,n);     /* Single vector of real coefficients */
data_I = dvector(1,n);    /* Single vector of imaginary cocoefficients */
averageR = dvector(1,n);  /* average of real components */
averageI = dvector(1,n);  /* average of imaginary components */
A_R = dmatrix(1,n,1,n);   /* covariance matrix of real vectors */
A_I = dmatrix(1,n,1,n);   /* covariance matrix of imaginary vectors */
V = dmatrix(1,N,1,N);     /* Matrix of eigenvectors */
W = dvector(1,N);         /* Matrix of eigenvalues */



/* OPEN AVERAGES FILES */

/* open file of Real averages */
outlhandle = fopen(averagesR,"r");
if (outlhandle == NULL)
{
      printf("Can't open file %s, Exiting to system\n",averagesR);
      exit(-1);
}

/* load real array */
for (i=1; i<=n; i++)
{
      fscanf(outlhandle,"%e",&tempfloat);
      averageR[i] = (double) tempfloat;
}
fclose(outlhandle);

/* open file of Imaginary averages */
outlhandle = fopen(averagesI,"r");
if (outlhandle == NULL)
{
      printf("Can't open file %s, Exiting to system\n",averagesI);
      exit(-1);
}
```

```
/* load imaginary array */
for (i=1; i<=n; i++)
{
      fscanf(outlhandle,"%e",&tempfloat);
      averageI[i] = (double) tempfloat;
}
fclose(outlhandle);


/* OPEN SPEECH DATA FILE */

inhandle = fopen(argv[1],"r");
if(inhandle == NULL)
{
      printf("Can't open file %s." ,argv[1]);
      exit(-1);
}

/* read entire file and find maximum amplitude */
New_vectors = integers_read = 0;
fscanf(inhandle,"%d",&MAX_AMP);
while(fscanf(inhandle,"%d",&tempint) != EOF)
{
      if (abs(MAX_AMP) < abs(tempint))
            MAX_AMP = tempint;
    integers_read++;
    if (integers_read == N)
    {
        New_vectors++;
        integers_read = 0;
    }
}
rewind(inhandle);


/* LOAD COVARIANCE MATRICES */

/* open Real covariance file */
if ((outlhandle = fopen(covarianceR,"r")) != NULL)
{
      /* file exists */
      fclose(outlhandle);
      outlhandle = fopen(covarianceR,"r+");
      if (outlhandle == NULL)
      {
            printf("Can't open file %s, Exiting to system\n",covarianceR);
            exit(-1);
      }
}
else
{
```

```
        /* create file for writing and reading */
        outlhandle = fopen(covarianceR,"w+");
        if (outlhandle == NULL)
        {
              printf("Can't open file %s, Exiting to system\n",covarianceR);
              exit(-1);
        }

        /* write zero covariances to file */
        for (i=1; i<=n; i++)
              for (j=1; j<=n; j++)
                   fprintf(outlhandle,"%e\n",(float) zero);

        /* reset pointer to start of file */
        rewind(outlhandle);
  }

  /* load covariance array */
  for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
        {
              fscanf(outlhandle,"%e",&tempfloat);
              A_R[i][j] = (double) tempfloat;
        }
  rewind(outlhandle);

  /* open Imaginary covariance file */
  if ((out2handle = fopen(covarianceI,"r")) != NULL)
  {
        /* file exists */
        fclose(out2handle);
        out2handle = fopen(covarianceI,"r+");
        if (out2handle == NULL)
        {
              printf("Can't open file %s, Exiting to system\n",covarianceI);
              exit(-1);
        }
  }
  else
  {
        /* create file for writing and reading */
        out2handle = fopen(covarianceI,"w+");
        if (out2handle == NULL)
        {
              printf("Can't open file %s, Exiting to system\n",covarianceI);
              exit(-1);
        }

        /* write zero covariances to file */
        for (i=1; i<=n; i++)
              for (j=1; j<=n; j++)
                   fprintf(out2handle,"%e\n",(float) zero);
```

```
      /* reset pointer to start of file */
      rewind(out2handle);
}

/* load covariance array */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(out2handle,"%e",&tempfloat);
            A_I[i][j] = (double) tempfloat;
      }
rewind(out2handle);



/* UPDATE COVARIANCE MATRICES */

/* two passes of file, 0% and 50% overlap */
for (x=1; x<=2; x++)
{
      if (x==2) /* offset pointer for 50% overlap */
            for (i=1; i<=n; i++)
                  fscanf(inhandle,"%d",&tempint);

      /* update covariance matrix with frequency vectors */
      integers_read = k = 0;  /* integer counter */
      while(k<New_vectors)
      {
            fscanf(inhandle,"%d",&tempint);
            integers_read++;
            buff[integers_read] = tempint;

            /* increment input counter if whole vector was read */
            if (integers_read == N)
            {
                  /* convert input characters to doubles */
                  for (i=1; i<=N; i++)
                  {
                        d[2*i-1] = (double) buff[i];
                        d[2*i-1] /= MAX_AMP;
                  }

                  /* compute complex fft */
                  four1(d,N,1); /* overwrite input array */

                  /* collect real and imaginary vectors, ignore DC */
                  for (i=1; i<=n; i++)
                  {
                        data_R[i] = d[2*i+1];
                        data_I[i] = d[2*i+2];
                  }
```

```
                        /* subtract average of each component */
                        for (i=1; i<=n; i++)
                        {
                              data_R[i] -= averageR[i];
                              data_I[i] -= averageI[i];
                        }

                        /* update covariance matrix */
                        for (i=1; i<=n; i++)
                              for (j=1; j<=n; j++)
                              {
                                    A_R[i][j] += data_R[i]*data_R[j];
                                    A_I[i][j] += data_I[i]*data_I[j];
                              }

                        /* clear input array */
                        for (i=1; i<=2*N; i++)
                              d[i]=0.0;

                        integers_read = 0; /* reset for next vector */
                        k++;
                  }
            }
            rewind(inhandle);
      }

      /* write covariances to file */
      for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
                  fprintf(out1handle,"%e\n",(float) A_R[i][j]);
      fclose(out1handle);
      for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
                  fprintf(out2handle,"%e\n",(float) A_I[i][j]);
      fclose(out2handle);
}
```

LISTING THREE

```
/*                KARHUNEN - LOEVE TRANSFORMATION PROGRAM

            Command Line Inputs : klt

      This program decomposes the two covariance matrices using Singular Value
 Decomposition.   The SVD routine was extracted from Numerical recipes in C
 (Cambridge Press).

      The eigenvectors of the covariance matrix overwrite the columns of the
 covariance matrix and written to files as the Karhunen-Loeve Transform matrices.

            OUTPUTS : File KLT_Re.dat
                    : File KLT_Im.dat

   Program written by FLTLT Don Dryley at AFIT, Sep 1992
*/



#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define MAX_AMP 32768
#define N 256
#define n 128


void main(argc,argv)
int argc;
char *argv[];
{
      FILE        *inhandle,*out1handle,*out2handle;
      int         i, j, k, x, vectors, integers_read, tempint;
      float       tempfloat;
      double      **A_R, **A_I, *W, **V;
      char        covarianceR[] = "covar_Re.dat",
                  covarianceI[] = "covar_Im.dat",
                  transformR[] = "KLT_Re.dat",
```

```
            transformI[] = "KLT_Im.dat";

 /* CREATE ARRAYS, INITIALISED TO ZERO */

 A_R = dmatrix(1,n,1,n);       /* covariance of real coefficients */
 A_I = dmatrix(1,n,1,n);       /* covariance of imaginary coefficients */
 V = dmatrix(1,n,1,n);         /* Matrix of eigenvectors */
 W = dvector(1,n);             /* Matrix of eigenvalues */


 /* LOAD COVARIANCE MATRICES */

 /* open Real covariance file */
 outlhandle = fopen(covarianceR,"r");
 if (outlhandle == NULL)
 {
       printf("Can't open file %s, Exiting to system\n",covarianceR);
       exit(-1);
 }

 /* load covariance array */
 for (i=1; i<=n; i++)
       for (j=1; j<=n; j++)
       {
             fscanf(outlhandle,"%e",&tempfloat);
             A_R[i][j] = (double) tempfloat;
       }
 fclose(outlhandle);

 /* open Imaginary covariance file */
 out2handle = fopen(covarianceI,"r");
 if (out2handle == NULL)
 {
       printf("Can't open file %s, Exiting to system\n",covarianceI);
       exit(-1);
 }

 /* load covariance array */
 for (i=1; i<=n; i++)
       for (j=1; j<=n; j++)
       {
             fscanf(out2handle,"%e",&tempfloat);
             A_I[i][j] = (double) tempfloat;
       }
 fclose(out2handle);


 /* FORM KL TRANSFORMS */

 /* find and sort eigenvectors of A_R and A_I */
 svdcmp(A_R,n,n,W,V);
```

```
        eigsrt(W,A_R,n);

        /* open KL_R transform file */
        outlhandle = fopen (transformR,"w");
        if(outlhandle == NULL)
        {     printf("Can't open file %s",transformR);
              exit(-1);
        }

        /* write real part of eigenvectors to KL_R transform file */
        for (i=1; i<=n; i++)
              for (j=1; j<=n; j++)
                    fprintf(outlhandle,"%e\n",(float) A_R[j][i]);
        fclose(outlhandle);


        svdcmp(A_I,n,n,W,V);
        eigsrt(W,A_I,n);

        /* open KL_I transform file */
        outlhandle = fopen (transformI,"w");
        if(outlhandle == NULL)
        {     printf("Can't open file %s",transformI),
              exit(-1);
        }

        /* write imaginary parts of eigenvectors to KL_I transform file */
        for (i=1; i<=n; i++)
              for (j=1; j<=n; j++)
                    fprintf(outlhandle,"%e\n",(float) A_I[j][i]);
        fclose(outlhandle);
}
```

LISTING FOUR

```
/*                          SPEECH REDUCTION PROGRAM

            Command Line Inputs : reduce speech.dat

      Speech source files are first converted from the NeXT sound format into a
 data format using the program sound_to.   Sound_to creates two files, a header
 file which contains the source file's header information and a data file of
 integers which between -32768 and 32768 containing the speech.

      This program uses the original sound file's header with the file size
 adjusted to the reconstructed file size (usually not same size as original).

      The program prompts the user for the number of coefficients used for the
 reduction experiment.

            Outputs: temp_2.dat

      Program written by FLTLT Don Dryley at AFIT Sep 92
*/


#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define MAX_AMP 32768
#define N 256
#define n 128


typedef struct
{
      int magic;
      int DataLocation;
      int DataSize;
      int DataFormat;
      int SamplingRate;
      int ChannelCount;
      char info[4];
} SNDheader;
```

```
void transform__(trans_R,trans_I,dim,in_R,in_I,size,KL_R,KL_I)
double *trans_R,*trans_I,*in_R,*in_I,**KL_R,**KL_I;
int dim,size;
/* function transforms two input vectors of dimension size */
/* into two vectors of coefficients of dimension dim using */
/* the transformation matrices KL_R and KL_I               */
{
      int i,j;

      /* transform = KL multiplied by in_vector */
      for (i=1; i<=dim; i++)
      {
            trans_R[i] = trans_I[i] = 0.0;
            for (j=1; j<=size; j++)
            {
                  trans_R[i] += KL_R[i][j] * in_R[j];
                  trans_I[i] += KL_I[i][j] * in_I[j];
            }
      }
}

void inverse_transform__(inv_R,inv_I,size,red_R,red_I,dim,KL_R,KL_I)
double *inv_R,*inv_I,*red_R,*red_I,**KL_R,**KL_I;
int size,dim;
/* function inverse transforms two vectors of coefficients of */
/* dimension dim into two output vectors of dimension size    */
/* using the transformation matrices KL_R and KL_I            */
{
      int i,j;

      /* inverse transform = KLI multiplied by coefficients */
      for (i=1; i<=size; i++)
      {
            inv_R[i] = inv_I[i] = 0.0;
            for (j=1; j<=dim; j++)
            {
                  inv_R[i] += KL_R[j][i] * red_R[j];
                  inv_I[i] += KL_I[j][i] * red_I[j];
            }
      }
}


void main(argc,argv)
int argc;
char *argv[];
{
      SNDheader   SND;
      FILE        *inhandle, *outhandle;
      short       *tempdata;
      int         i, j, k, vectors, integers_read, tempint, dim;
      long        *buff, templong;
```

```
float       tempfloat;
double      **KL_R, **KL_I, *averageR, *averageI, *d, *org_R, *org_I,
            *rec_R, *rec_I,*reduce_R, *reduce_I, DC, magnitude;
char        KLT_R[] = "KLT_Re.dat",
            KLT_I[] = "KLT_Im.dat",
            AVG_R[] = "avgs_Re.dat",
            AVG_I[] = "avgs_Im.dat",
            temp[] = "temp_2.dat";


/* CHECK ARGUMENTS */

if( argc != 2)
{     printf("Format: D>reduce speech.dat ");
      exit(-1);
}


/* PROMPT FOR NUMBER OF KL COEFFICIENTS */

printf("Enter number of coefficients <1 - %d> ... ",n);
scanf("%d",&dim);
printf("\n\n");


/* CREATE MATRICES */

buff = lvector(1,N);          /* input buffer */
d = dvector(1,2*N);           /* FFT of one frame of speech is complex */
averageR = dvector(1,n);      /* average of discrete frequencies */
averageI = dvector(1,n);      /* average of discrete frequencies */
KL_R = dmatrix(1,n,1,n);      /* hold KL transform */
KL_I = dmatrix(1,n,1,n);      /* hold KL transform */
org_R = dvector(1,n);         /* holds original frequency vector */
org_I = dvector(1,n);         /* holds original frequency vector */
rec_R = dvector(1,n);         /* holds reconstructed frequency vector */
rec_I = dvector(1,n);         /* holds original frequency vector */
reduce_R = dvector(1,n);      /* holds set of coefficients */
reduce_I = dvector(1,n);      /* holds original frequency vector */


/* READ TRANSFORM FROM FILE */

/* open Real transform file */
inhandle = fopen(KLT_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_R);
      exit(-1);
}
```

```
/* load KL from transform file */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(inhandle,"%e",&tempfloat);
            KL_R[i][j] = (double) tempfloat;
      }
fclose(inhandle);

/* open Imaginary transform file */
inhandle = fopen(KLT_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_I);
      exit(-1);
}

/* load KL from transform file */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(inhandle,"%e",&tempfloat);
            KL_I[i][j] = (double) tempfloat;
      }
fclose(inhandle);



/* READ FREQUENCY AVERAGES FROM FILE */

/* open real averages file */
inhandle = fopen(AVG_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_R);
      exit(-1);
}

/* load averages from averages file */
for (j=1; j<=n; j++)
{
      fscanf(inhandle,"%e",&tempfloat);
      averageR[j] = (double) tempfloat;
}
fclose(inhandle);

/* open imaginary averages file */
inhandle = fopen(AVG_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_I);
      exit(-1);
}
```

```
/* load averages from averages file */
for (j=1; j<=n; j++)
{
      fscanf(inhandle,"%e",&tempfloat);
      averageI[j] = (double) tempfloat;
}
fclose(inhandle);


/* KL TRANSFORMATION EXPERIMENT */

/* open temporary file for data */
outhandle = fopen(temp,"w");
if (outhandle == NULL)
{     printf("Can't open file %s.",temp);
      exit(-1);
}

/* open source file for reading */
inhandle = fopen(argv[1],"r");
if (inhandle == NULL)
{     printf("Can't open file %s.",argv[1]);
      exit(-1);
}

/* count number of vectors in source file */
integers_read = vectors = 0;
while (fscanf(inhandle,"%d",&tempint) != EOF)
{
      integers_read++;
      if (integers_read == N)
      {
            integers_read = 0;
            vectors++;
      }
}
rewind(inhandle);


/* KL TRANSFORM SPEECH */

integers_read = 0;
j = 1;
while (j <= vectors)
{
      fscanf(inhandle,"%d",&tempint);
      integers_read++;
      buff[integers_read] = tempint;
```

```
/* transform a vector */
if (integers_read == N)
{
      /* load input array to fourier transform */
      for (k=1; k<=N; k++)
      {
           d[2*k-1] = (double) buff[k];
           d[2*k-1] /= MAX_AMP;
      }

      /* overwrite input with complex frequencies */
      four1(d,N,1);

      /* extract real, imaginary and DC components */
      rect(d,n,org_R,org_I,&DC);

      /* subtract mean from each component */
      for (k=1; k<=n; k++)
      {
           org_R[k] -= averageR[k];
           org_I[k] -= averageI[k];
      }

      /* generate KL coefficients */
      transform__(reduce_R,reduce_I,dim,org_R,org_I,n,KL_R,KL_I);

      /* reconstruct frequency vector from KL coefficients */
      inverse_transform__(rec_R,rec_I,n,reduce_R,reduce_I,dim,KL_R,
      KL_I);

      /* clear input array for inverse FFT */
      for (k=1; k<=2*N; k++)
           d[k] = 0.0;

      /* insert DC value */
      d[1] = DC;

      /* load input array with reconstructed spectrum */
      for (k=2; k<=n+1; k++)
      {
           d[2*k-1] = rec_R[k-1] + averageR[k-1];
           d[2*k] = rec_I[k-1] + averageI[k-1];
      }
      for (k=2; k<=n; k++)
      {
           d[N+2*k-1] = rec_R[n-k+1] + averageR[n-k+1];
           d[N+2*k] = -1.0 * (rec_I[n-k+1] + averageI[n-k+1]);
      }

      /* reconstruct speech from reconstructed vector */
      four1(d,N,-1);
```

```
                    /* write reconstucted time domain data to output file */
                    for (k=1; k<=N; k++)
                    {
                         d[2*k-1] *= MAX_AMP;
                         d[2*k-1] /= N;
                         fprintf(outhandle,"%d\n",(int) d[2*k-1]);
                    }

                    /* clear input array for next vector */
                    for (k=1; k<=2*N; k++)
                         d[k] = 0.0;

                    /* increment vector counter */
                    j++;

                    /* reset elements per vector counter */
                    integers_read=0;
               }
          }
          fclose(inhandle);
          fclose(outhandle);
}
```

LISTING FIVE

```
/* This is the header file for the routines declared in recipes.c */

typedef struct FCOMPLEX
{
      double r,i;
} fcomplex;

void nrerror();
int *ivector();
long *lvector();
float *vector();
double *dvector();
float **matrix();
double **dmatrix();
void free_ivector();
void free_lvector();
void free_vetcor();
void free_dvector();
void free_matrix();
void free_dmatrix();
fcomplex Complex();
fcomplex Cadd();
fcomplex Csub();
fcomplex Cdiv();
fcomplex Cmul();
fcomplex Cdiv();
fcomplex Conjg();
fcomplex Cinv();
double Cabs();
fcomplex Csqrt();
fcomplex RCmul();
void four1();
void mag();
void rect();
void eigsrt();
void svdcmp();
```

LISTING SIX

```
/* This file contains routines from Numerical recipes in C */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>



typedef struct FCOMPLEX
{
      double r,i;
} fcomplex;


#define ROTATE(a,i,j,k,l) g=a[i][j];h=a[k][l];a[i][j]=g-s*(h+g*tau);\
      a[k][l]=h+s*(g-h*tau);


#define SWAP(a,b) tempr=(a); (a)=(b); (b)=tempr


/*    PYTHAG computes sqrt(sqr(a)+sqr(b)) without    */
/*        destructive overflow or underflow          */
static float at, bt, ct;
#define PYTHAG(a,b)     ( ( a t = f a b s ( a ) )    >    ( b t = f a b s ( b ) )    ?
                        \(ct=bt/at,at*sqrt(1.0+ct*ct))    :    (bt    ?
                        (ct=at/bt,bt*sqrt(1.0+ct*ct)) : 0.0))


static float maxarg1,maxarg2;
#define MAX(a,b)  (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ?\ (maxarg1) :
                  (maxarg2))


#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))



void nrerror(error_text)
char error_text[];
/* Numerical Recipes standard error handler */
{
      fprintf(stderr,"Numerical recipes run-time error...\n");
      fprintf(stderr,"%s\n", error_text);
      fprintf(stderr,"...now exiting to system...\n");
      exit(1);
}
```

```
int *ivector(nl,nh)
int nl, nh;
/* Allocates a vector of integers from nl to nh */
{
      int *v,i;

      v = (int *) malloc((unsigned) (nh-nl+1)*sizeof(int));

      if (!v) nrerror("allocation failure in ivector()");
      else for (i=nl; i<=nh; i++)
            v[i] = 0;

      return v-nl;
}



long *lvector(nl,nh)
int nl,nh;
/* allocates a long int vector with range [nl..nh] */
{
      long *v;
      int i;

      v=(long *)malloc((unsigned) (nh-nl+1)*sizeof(long));
      if (!v) nrerror("allocation failure in ivecto()");
      else for (i=nl; i<=nh; i++)
            v[i] = 0;

      return v-nl;
}



float *vector(nl,nh)
int nl,nh;
/* Allocates a float vector with range [nl..nh] */
{
      float *v;
      int i;

      v=(float *)malloc((unsigned) (nh-nl+1)*sizeof(float));
      if (!v) nrerror("allocation failure in vector()");
      else for (i=nl; i<=nh; i++)
            v[i] = 0;
      return v-nl;
}
```

```
double *dvector(nl,nh)
int nl, nh;
/* Allocates a double vector with range [nl..nh] */
{
      double *v;
      int i;

      /* Allocate pointers to rows */
      v=(double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
      if (!v) nrerror("Allocation failure dvector()");
      else for (i=nl; i<=nh; i++)
            v[i] = 0;
      return v - nl;
}



float **matrix(nrl,nrh,ncl,nch)
int nrl, nrh, ncl, nch;
/* Allocates a float matrix with range [nrl..nrh][ncl..nch] */
{
      int i,j;
      float **m;

      /* Allocate pointers to rows */
      m=(float **) malloc((unsigned) (nrh-nrl+1)*sizeof(float*));
      if (!m) nrerror("Allocation failure 1 in matrix()");
      m -= nrl;

      /* Allocate rows and set pointers to them */
      for(i=nrl; i<=nrh; i++)
      {     m[i]=(float *) malloc((unsigned) (nch-ncl+1)*sizeof(float));
            if (!m[i]) nrerror("Allocation failure 2 in matrix()");
            m[i] -= ncl;
      }

      for (i=nrl; i<=nrh; i++)
            for (j=ncl; j<=nch; j++)
                  m[i][j] = 0.0;

      return m;
}



double **dmatrix(nrl,nrh,ncl,nch)
int nrl, nrh, ncl, nch;
/* Allocates a double matrix with range [nrl..nrh][ncl..nch] */
{
      int i,j;
      double **m;
```

```
        /* Allocate pointers to rows */
        m=(double **) malloc((unsigned) (nrh-nrl+1)*sizeof(double*));
        if (!m) nrerror("Allocation failure 1 in dmatrix()");
        m -= nrl;

        /* Allocate rows and set pointers to them */
        for(i=nrl; i<=nrh; i++)
        {       m[i]=(double *) malloc((unsigned) (nch-ncl+1)*sizeof(double));
                if (!m[i]) nrerror("Allocation failure 2 in dmatrix()");
                m[i] -= ncl;
        }

        for (i=nrl; i<=nrh; i++)
                for (j=ncl; j<=nch; j++)
                        m[i][j] = 0.0;

        return m;
}



void free_ivector(v,nl)
int *v, nl;
/* Frees an integer vector allocated by ivector() */
{
        free((char*) (v+nl));
}



void free_lvector(v,nl)
long *v;
int nl;
/* Frees a vector of long allocated by lvector() */
{
        free((char*) (v+nl));
}



void free_vector(v,nl)
float *v;
int nl;
/* Frees a float vector allocated by vector() */
{
        free((char*) (v+nl));
}
```

```
void free_dvecto    nl)
double *v;
int nl;
/* Frees a double vector allocated by dvector() */
{
      free((char*) (v+nl));
}



void free_matrix(m,nrl,nrh,ncl)
float **m;
int nrl,nrh,ncl;
/* Frees a matrix allocated with matrix() */
{
      int i;

      for (i=nrh; i>=nrl; i--) free((char*) (m[i]+ncl));
            free((char*) (m+nrl));
}



void free_dmatrix(m,nrl,nrh,ncl)
double **m;
int nrl,nrh,ncl;
/* Frees a matrix allocated with dmatrix() */
{
      int i;

      for (i=nrh; i>=nrl; i--) free((char*) (m[i]+ncl));
            free((char*) (m+nrl));
}



fcomplex Complex(re,im)
double re,im;
/* Returns a complex number with specified real and imaginary parts */
{
      fcomplex c;

      c.r = re;
      c.i = im;

      return c;
}
```

```
fcomplex Cadd(a,b)
fcomplex a,b;
/* Returns the complex sum of two complex numbers */
{
      fcomplex c;

      c.r = a.r + b.r;
      c.i = a.i + b.i;

      return c;
}



fcomplex Csub(a,b)
fcomplex a,b;
/* Returns the complex difference of two complex numbers */
{
      fcomplex c;

      c.r = a.r - b.r;
      c.i = a.i - b.i;

      return c;
}



fcomplex Cmul(a,b)
fcomplex a,b;
/* Returns the complex product of two complex numbers */
{
      fcomplex c;

      c.r = a.r*b.r - a.i*b.i;
      c.i = a.i*b.r + a.r*b.i;

      return c;
}
```

```
fcomplex Cdiv(a,b)
fcomplex a,b;
/* Returns the complex quotient of two complex numbers */
{
      fcomplex c;
      double r,den;

      if (fabs(b.r) >= fabs(b.i))
      {
            r = b.i/b.r;
            den = b.r + r*b.i;
            c.r = (a.r + r*a.i)/den;
            c.i = (a.i - r*a.r)/den;
      }
      else
      {
            r = b.r/b.i;
            den = b.i + r*b.r;
            c.r = (a.r*r + a.i)/den;
            c.i = (a.i*r - a.r)/den;
      }

      return c;
}



fcomplex Conjg(z)
fcomplex z;
/* Returns the conjugate of a complex number */
{
      fcomplex c;

      c.r = z.r;
      c.i = -1.0*z.i;

      return c;
}



fcomplex Cinv(z)
fcomplex z;
/* Returns the inverse of a complex number */
{
      fcomplex c;

      c =  Cdiv(Conjg(z),Cmul(z,Conjg(z)));

      return c;
}
```

```
double Cabs(z)
fcomplex z;
/* Returns the absolute value of a complex number */
{
      double x, y, ans, temp;

      x = fabs(z.r);
      y = fabs(z.i);
      if (x == 0.0)
            ans = y;
      else
            if (y == 0.0)
                  ans = x;
            else
                  if (x > y)
                  {
                        temp = y / x;
                        ans = x * sqrt(1.0 + temp * temp);
                  }
                  else
                  {
                        temp = x / y;
                        ans = y * sqrt(1.0 + temp * temp);
                  }

      return ans;
}



fcomplex Csqrt(z)
fcomplex z;
/* Returns the complex square root of a complex number */
{
      fcomplex c;
      double x, y, w, r;

      if ((z.r == 0.0) &&  (z.i == 0.0))
            c.r = c.i = 0.0;
      else
      {
            x = fabs(z.r);
            y = fabs(z.i);
            if (x >= y)
            {
                  r = y / x;
                  w = sqrt(x) * sqrt(0.5 * (1.0 + sqrt(1.0 + r * r)));
            }
            else
            {
                  r = x / y;
                  w = sqrt(y) * sqrt(0.5 * (r + sqrt(1.0 + r * r)));
            }
```

```
            if (z.r >= 0.0)
            {
                  c.r = w;
                  c.i = z.i / (2.0 * w);
            }
            else
            {
                  c.i = (z.i >= 0) ? w : -w;
                  c.r = z.i/(2.0 * c.i);
            }
      }

      return c;
}



fcomplex RCmul(x,a)
double x;
fcomplex a;
/* Returns the complex product of a real number and a complex number */
{
      fcomplex c;

      c.r = x * a.r;
      c.i = x * a.i;

      return c;
}



void four1(data_in,nn,isign)
double *data_in;
int nn,isign;
/*    This function replaces data_in with the complex FFT */
/* if isign is 1, or with the inverse FFT if isign is -1. */
/* nn is the number of time samples in the input frame    */
{
      int n, mmax, m, j, istep, i;
      double wtemp, wr, wpr, wpi, wi, theta;
      double tempr, tempi;
      n=nn << 1;
      j=1;
      for (i=1; i<n; i+=2)
      {
            if (j>i)
            {
                  SWAP(data_in[j],data_in[i]);
                  SWAP(data_in[j+1],data_in[i+1]);
            }
            m=n >> 1;
```

```
        while (m >= 2 && j > m)
        {
            j -= m;
            m >>= 1;
        }
        j += m;
    }
    mmax=2;
    while (n> mmax)
    {
        istep=2*mmax;
        theta=6.28318530717959/(isign*mmax);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (m=1; m<mmax; m+=2)
        {
            for (i=m; i<=n; i +=istep)
            {
                j=i+mmax;
                tempr=wr*data_in[j]-wi*data_in[j+1];
                tempi=wr*data_in[j+1]+wi*data_in[j];
                data_in[j]=data_in[i]-tempr;
                data_in[j+1]=data_in[i+1]-tempi;
                data_in[i] += tempr;
                data_in[i+1] += tempi;
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        mmax=istep;
    }
}
```

```
void mag(rect_data,length,mag_data,DC)
double *rect_data, *mag_data;
int length, DC;
/* Rect_data is the input of 'length' rectangular complex pairs R + jI. */
/* The magnitude of each pair, mag_data[j], is the output               */
{
      int i;
      double temp;

      for (i=1; i<=length+1; i++)
      {
            temp = sqrt(pow(rect_data[2*i-1],2) + pow(rect_data[2*i],2));
            if (i==1)
                  DC = temp;
            else
                  mag_data[i-1] = temp;
      }
}



void rect(rect_data,length,data_r, data_i,DC)
double *rect_data, *data_r, *data_i, *DC;
int length;
/* Rect_data is the input of 'length' rectangular complex pairs R + jI.     */
/* The arrays data_r and data_i are the respective real and imag components */
{
      int i;
      double temp;

      *DC = rect_data[1];

      for (i=2; i<=length+1; i++)
      {
            data_r[i-1] = rect_data[2*i-1];
            data_i[i-1] = rect_data[2*i];
      }
}



void eigsrt(d,v,n)
double *d, **v;
int n;
{
      int k,j,i;
      double p;

      for (i=1;i<n;i++)
      {
            p=d[k=i];
            for (j=i+1;j<=n;j++)
                  if (d[j] >= p) p=d[k=j];
```

```
            if (k != i)
            {
                  d[k]=d[i];
                  d[i]=p;
                  for (j=1;j<=n;j++)
                  {
                        p=v[j][i];
                        v[j][i]=v[j][k];
                        v[j][k]=p;
                  }
            }
      }
}


void svdcmp(a,m,n,w,v)
double **a, *w, **v;
int m, n;
/* Given a matrix a[1..m,1..n], this routine computes its singular value   */
/* decomposition, A = U.W.Vt.   The Matrix U replaces [a] on output.   The */
/* diagonal matrix of singular values W is output as a vector w[1..n].     */
/* The matrix V (not the transpose of V, Vt) is output as v[1..n,1..n].    */
/* m must be greater than or equal to n; if it is smaller, then A should   */
/* be filled up to square with zero rows                                   */
{
      int flag, i, its, j, jj, k, l, nm;
      double c, f, h, s, x, y, z;
      double anorm=0.0, g=0.0, scale=0.0;
      double *rv1, *dvector();
      void nrerror(), free_vector();

      if (m < n) nrerror("SVDCMP: You must augment with extra zero rows");
      rv1=dvector(1,n);

      /* Householder reduction to bidiagonal form */
      printf("Householder reduction started\n");
      for (i=1; i<=n; i++)
      {
            l=i+1;
            rv1[i]=scale*g;
            g=s=scale=0.0;
            if (i <= m)
            {
                  for (k=i; k<=m; k++) scale += fabs(a[k][i]);
                  if (scale)
                  {
                        for (k=i; k<=m; k++)
                        {
                              a[k][i] /= scale;
                              s += a[k][i]*a[k][i];
                        }
                        f=a[i][i];
```

```
                        g = -SIGN(sqrt(s),f);
                        h=f*g-s;
                        a[i][i]=f-g;
                        if (i != n)
                        {
                              for (j=l; j<=n; j++)
                              {
                                    for (s=0.0,k=i; k<=m; k++)
                                          s += a[k][i]*a[k][j];
                                    f=s/h;
                                    for (k=i; k<=m; k++) a[k][j] += f*a[k][i];
                              }
                        }
                        for (k=i; k<=m; k++) a[k][i] *= scale;
                  }
            }
            w[i]=scale*g;
            g=s=scale=0.0;
            if (i <= m && i != n)
            {
                  for (k=l; k<=n; k++) scale += fabs(a[i][k]);
                  if (scale)
                  {
                        for (k=l; k<=n; k++)
                        {
                              a[i][k] /= scale;
                              s += a[i][k]*a[i][k];
                        }
                        f=a[i][l];
                        g = -SIGN(sqrt(s),f);
                        h=f*g-s;
                        a[i][l]=f-g;
                        for (k=l; k<=n; k++) rvl[k]=a[i][k]/h;
                        if (i != m)
                        {
                              for (j=l; j<=m; j++)
                              {
                              for (s=0.0,k=l; k<=n; k++)
                                    s += a[j][k]*a[i][k];
                              for (k=l; k<=n; k++)
                                    a[j][k] += s*rvl[k];
                              }
                        }
                        for (k=l; k<=n; k++) a[i][k] *= scale;
                  }
            }
            anorm=MAX(anorm,(fabs(w[i])+fabs(rvl[i])));
      }
```

```
/* Accumulation of right-hand transformation */
for (i=n; i>=1; i--)
{
      if (i<n)
      {
            if (g)
            {
                  for (j=l; j<=n; j++)
                        v[j][i]=(a[i][j]/a[i][l])/g;
                  for (j=l; j<=n; j++)
                  {
                        for (s=0.0,k=l; k<=n; k++)
                              s += a[i][k]*v[k][j];
                        for (k=l; k<=n; k++)
                              v[k][j] += s*v[k][i];
                  }
            }
            for (j=l; j<=n; j++) v[i][j]=v[j][i]=0.0;
      }
      v[i][i]=1.0;
      g=rvl[i];
      l=i;
}

/* Accumulation of left-hand transformations */
for (i=n; i>=1; i--)
{
      l=i+1;
      g=w[i];
      if (i < n)
            for (j=l; j<=n; j++) a[i][j]=0.0;
      if (g)
      {
            g=1.0/g;
            if (i != n)
            {
                  for (j=l; j<=n; j++)
                  {
                        for (s=0.0,k=l; k<=m; k++)
                              s += a[k][i]*a[k][j];
                        f=(s/a[i][i])*g;
                        for (k=i; k<=m; k++)
                              a[k][j] += f*a[k][i];
                  }
            }
            for (j=i; j<=m; j++)
                  a[j][i] *= g;
      }
      else
            for (j=i; j<=m; j++)
                  a[j][i] = 0.0;
      ++a[i][i];
}
```

```
/* Diagonalization of the bidiagonal form */
printf("Diagonalization started\n");
for (k=n; k>=1; k--)                    /*  loop over singular values   */
{
      for (its=1; its<=30; its++)    /* loop over allowed iterations */
      {
            flag=1;
            for (l=k; l>=1; l--)    /* test for splitting */
            {
                  nm=l-1;           /* note that rv[1] is always zero */
                  if ((double)(fabs(rv1[l])+anorm) == anorm)
                  {
                        flag=0;
                        break;
                  }
                  if ((double)(fabs(w[nm])+anorm) == anorm) break;
            }
            if (flag)
            {
                  c=0.0;    /* cancellation of rv1[l], if l>1 */
                  s=1.0;
                  for (i=l; i<=k; i++)
                  {
                        f=s*rv1[i];
                        rv1[i]=c*rv1[i];
                        if ((double)(fabs(f)+anorm) == anorm) break;
                        g=w[i];
                        h=PYTHAG(f,g);
                        w[i]=h;
                        h=1.0/h;
                        c=g*h;
                        s=(-f*h);
                        for (j=1; j<=m; j++)
                        {
                              y=a[j][nm];
                              z=a[j][i];
                              a[j][nm]=y*c+z*s;
                              a[j][i]=z*c-y*s;
                        }
                  }
            }
            z=w[k];
            if (l == k)         /* convergence */
            {
                  if (z < 0.0)  /* Singular value is made non-negative */
                  {
                        w[k] = -z;
                        for (j=1; j<=n; j++)
                              v[j][k]=(-v[j][k]);
                  }
                  break;
            }
```

```
if (its == 100)
      nrerror("No convergence in 100 SVDCMP iterations");
x=w[l];          /* Shift from bottom 2-by-2 minor */
nm=k-1;
y=w[nm];
g=rvl[nm];
h=rvl[k];
f=((y-z)*(y+z)+(g-h)*(g+h))/(2.0*h*y);
g=PYTHAG(f,1.0);
f=((x-z)*(x+z)+h*((y/(f+SIGN(g,f)))-h))/x;

/* Next QR information */
c=s=1.0;
for (j=l; j<=nm; j++)
{
      i=j+1;
      g=rvl[i];
      y=w[i];
      h=s*g;
      g=c*g;
      z=PYTHAG(f,h);
      rvl[j]=z;
      c=f/z;
      s=h/z;
      f=x*c+g*s;
      g=g*c-x*s;
      h=y*s;
      y=y*c;
      for (jj=1; jj<=n; jj++)
      {
            x=v[jj][j];
            z=v[jj][i];
            v[jj][j]=x*c+z*s;
            v[jj][i]=z*c-x*s;
      }
      z=PYTHAG(f,h);
      w[j]=z;
      if (z)      /* Rotation can be arbitrary if z=0 */
      {
            z=1.0/z;
            c=f*z;
            s=h*z;
      }
      f=(c*g)+(s*y);
      x=(c*y)-(s*g);
      for (jj=1; jj<=m; jj++)
      {
            y=a[jj][j];
            z=a[jj][i];
            a[jj][j]=y*c+z*s;
            a[jj][i]=z*c-y*s;
      }
}
```

```
                    rvl[1]=0.0;
                    rvl[k]=f;
                    w[k]=x;
                }
            }

            free_dvector(rvl,1);
        }
```

LISTING SEVEN

```
/*                      DATA FILE CONVERSION PROGRAM

          Command Line INPUTS :   data_to infile.dat infile.hed outfile.snd

     This program will input a specified datafile of ASCII integers and convert
into bytes which represent the sound.  An associated file containing the sound
file's header information is needed.

          OUTPUTS are: infile.snd

     Program adapted by FLTLT Don Dryley from a similar program written by CAPT
Jim Geurts.
*/


#include <stdio.h>


typedef struct
{
    int magic;          /* must be equal to SND_MAGIC */
    int dataLocation;   /* Offset or pointer to the raw data */
    int dataSize; /* Number of bytes of data in the raw data */
    int dataFormat;     /* The data format code */
    int samplingRate;   /* The sampling rate */
    int channelCount;   /* The number of channels */
   char info[4]; /* Textual information relating to the sound. */
} SNDSoundStruct;


main(argc,argv)
int argc;
char *argv[];
{
     SNDSoundStruct  snd;
     FILE            *infile, *outfile;
     short int       *ind;
     int             n,number_of_samples, temp;
```

```
/* CHECK COMMAND LINE ARGUMENTS */

if (argc != 4)
{
      printf("\n Incorrect Command Format \n Use data_snd infilename.dat
      infile.hed  outfilename.snd\n");
      exit(-1);
}



/* LOAD SOUND FILE HEADER FROM HEADER FILE */

/* open header file */
infile = fopen(argv[2],"r");
if (infile == NULL)
{
      printf("Cannot open %s, Exiting to system\n",argv[2]);
      exit(-1);
}

/*  Read in the header information */
fread(&snd,sizeof(SNDSoundStruct),1,infile);

fclose(infile);



/* HOW MANY INTEGERS ARE THERE IN FILE ? */

/* open input data file */
infile = fopen(argv[1],"r");
if (infile == NULL)
{
      printf("Cannot open %s, Exiting to system\n",argv[1]);
      exit(-1);
}

/* count integers */
number_of_samples = 0;
while ( fscanf(infile,"%d",&temp) != EOF) number_of_samples++;



/* READ DATA FROM FILE INTO AN ARRAY */

/* set number of bytes to be read */
snd.dataSize = number_of_samples * sizeof(short);
```

```
        /* create array */
        ind = (short int *)malloc((unsigned) snd.dataSize);
        if (!ind)
        {
              printf("allocation failure, Exiting to system\n");
              exit(-1);
        }

        /* read data from file */
        rewind(infile);
        for (n=1; n<=number_of_samples; n++)
        {
              fscanf(infile,"%d",&temp);
              ind[n] = (short) temp;
        }

        fclose(infile);



        /* WRITE DATA IN SOUND FORMAT TO SOUND FILE */

        /* open output sound file */
        outfile = fopen(argv[3],"w");
        if (infile == NULL)
        {
              printf("Cannot open %s, Exiting to system\n",argv[3]);
              exit(-1);
        }

        /* write sound header */
        fwrite(&snd, sizeof(SNDSoundStruct),1,outfile);

        /* write data to sound file */
        fwrite(ind, sizeof(char),snd.dataSize,outfile);

        fclose(outfile);
}
```

LISTING EIGHT

```
/*                      SOUND FILE CONVERSION PROGRAM

            Command Line INPUTS :   sound_to   infilename.snd  outfilename.hed
                                    outfilename.dat

     This program will input a specified soundfile and output the ASCII integers
which represent the sound.  In addition, an associated file containing the sound
file's header information is created.


            OUTPUTS are :     speech.dat
                        :     speech.hed


     Program adapted by FLTLT Don Dryley from a similar program written by CAPT
Jim Geurts.
*/



#include <stdio.h>


typedef struct
{
    int magic;          /* must be equal to SND_MAGIC */
    int dataLocation;   /* Offset or pointer to the raw data */
    int dataSize; /* Number of bytes of data in the raw data */
    int dataFormat;     /* The data format code */
    int samplingRate;   /* The sampling rate */
    int channelCount;   /* The number of channels */
   char info[4]; /* Textual information relating to the sound. */
} SNDSoundStruct;



main(argc,argv)
int argc;
char *argv[];
{
     SNDSoundStruct  snd;
     FILE            *sndfile, *output_file;
     short int       *dat;
     int             n,number_of_samples;
     char            header[] = "snd_head.dat";
```

```
        /* Double-checks that an input and output file have been specified */
        if (argc != 4)
        {
              printf("\n Incorrect Command Format \n Use sound_to infile.snd
                     outfile.hed outfile.dat\n");
              exit(-1);
        }

        /*open sound file */
        sndfile = fopen(argv[1],"r");
        if (sndfile == NULL)
        {
              printf("Cannot open %s, Exiting to system\n",argv[1]);
              exit(-1);
        }

        /*  Read in the header information */
        fread(&snd,sizeof(SNDSoundStruct),1,sndfile);

        /* Open file for header information, and save header structure */
        output_file = fopen(argv[2],"w");
        if (output_file == NULL)
        {
              printf("Cannot open %s, Exiting to system\n",argv[2]);
              exit(-1);
        }
        fwrite(&snd, sizeof(SNDSoundStruct), 1, output_file);
        fclose(output_file);

        /*  Read in the digitized data from sound file */
        number_of_samples=snd.dataSize/2;
        dat=(short int *)malloc(number_of_samples*sizeof(short int));
        fseek(sndfile,snd.dataLocation,0);
        fread( dat, sizeof(short int), number_of_samples, sndfile);

        /* Open output file and output the data in ASCII form */
        output_file = fopen (argv[3],"w");
        if (output_file == NULL)
        {
              printf("Cannot open %s, Exiting to system\n",argv[3]);
              exit(-1);
        }
        for (n=0; n<number_of_samples; n++)
              fprintf(output_file,"%d\n",dat[n]);
        fclose(output_file);
}
```

APPENDIX D

```
/*                          MEAN SQUARE ERROR PROGRAM

            Command Line Inputs : reduce speech.dat

      Speech source files are first converted from the NeXT sound format into a
data format using the program sound_to.   Sound_to creates two files, a header
file which contains the source file's header information and a data file of
integers between -32768 and 32768.

      This program uses the original sound file's header with the file size
adjusted to the reconstructed file's size (usually not same size as original).

      The program prompts the user for the number of coefficients used for the
reduction experiment writes this value to an error file along with the MSE.


            Outputs : temp_2.dat
                    : error_2.dat


      Program written by FLTLT Don Dryley at AFIT Sep 92.
*/



#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"



#define MAX_AMP 32768
#define N 256
#define n 128



typedef struct
{
      int magic;
      int DataLocation;
      int DataSize;
      int DataFormat;
      int SamplingRate;
      int ChannelCount;
      char info[4];
} SNDheader;
```

```
void transform__(trans_R,trans_I,dim,in_R,in_I,size,KL_R,KL_I)
double *trans_R,*trans_I,*in_R,*in_I,**KL_R,**KL_I;
int dim,size;
/* function transforms two input vectors of dimension size */
/* into two vectors of coefficients of dimension dim using */
/* the transformation matrices KL_R and KL_I               */
{
      int i,j;

      /* transform = KL multiplied by in_vector */
      for (i=1; i<=dim; i++)
      {
            trans_R[i] = trans_I[i] = 0.0;
            for (j=1; j<=size; j++)
            {
                  trans_R[i] += KL_R[i][j] * in_R[j];
                  trans_I[i] += KL_I[i][j] * in_I[j];
            }
      }
}



void inverse_transform__(inv_R,inv_I,size,red_R,red_I,dim,KL_R,KL_I)
double *inv_R,*inv_I,*red_R,*red_I,**KL_R,**KL_I;
int size,dim;
/* function inverse transforms two vectors of coefficients of */
/* dimension dim into two output vectors of dimension size    */
/* using the transformation matrices KL_R and KL_I            */
{
      int i,j;

      /* inverse transform = KLI multiplied by coefficients */
      for (i=1; i<=size; i++)
      {
            inv_R[i] = inv_I[i] = 0.0;
            for (j=1; j<=dim; j++)
            {
                  inv_R[i] += KL_R[j][i] * red_R[j];
                  inv_I[i] += KL_I[j][i] * red_I[j];
            }
      }
}



void error_spectrum_(size,org_R,org_I,rec_R,rec_I,error_R,error_I)
double *org_R, *org_I, *rec_R, *rec_I, *error_R, *error_I;
int size;
/* function generates a normalised error spectrum by, for each */
/* frequency, dividing the difference between the original and */
/* reconstructed spectrums by the original spectrum.   Assume  */
/* assume that the original, reconstructed, and error vectors  */
```

```
/* are created prior to execution with the function vector     */
{
      int i;

      for (i=1; i<=size; i++)
      {
            error_R[i] = rec_R[i] - org_R[i];
            error_I[i] = rec_I[i] - org_I[i];
      }
}



void main(argc,argv)
int argc;
char *argv[];
{
      SNDheader   SND;
      FILE        *inhandle, *outhandle;
      short       *tempdata;
      int         i, j, k, vectors, integers_read, tempint, dim;
      long        *buff, templong;
      float       tempfloat;
      double      **KL_R, **KL_I, *averageR, *averageI, *d, *org_R, *org_I,
                  *rec_R, *rec_I, *reduce_R, *reduce_I, DC, magnitude, *ferr_R,
                  *ferr_I, *onetimeR, *onetimeI;
      char        KLT_R[] = "KLT_Re.dat",
                  KLT_I[] = "KLT_Im.dat",
                  AVG_R[] = "avgs_Re.dat",
                  AVG_I[] = "avgs_Im.dat",
                  temp[] = "temp_2.dat",
                  error[] = "error_2.dat";



      /* CHECK ARGUMENTS */

      if( argc != 2)
      {     printf("Format: D>reduce speech.dat ");
            exit(-1);
      }



      /* PROMPT FOR NUMBER OF KL COEFFICIENTS */

      printf("Enter number of coefficients <1 - %d> ... ",n);
      scanf("%d",&dim);
      printf("\n\n");
```

```
/* CREATE MATRICES */

buff = lvector(1,N);          /* input buffer */
d = dvector(1,2*N);           /* FFT of one frame of speech is complex */
averageR = dvector(1,n);      /* average of discrete frequencies */
averageI = dvector(1,n);      /* average of discrete frequencies */
KL_R = dmatrix(1,n,1,n);      /* hold KL transform */
KL_I = dmatrix(1,n,1,n);      /* hold KL transform */
org_R = dvector(1,n);         /* holds original frequency vector */
org_I = dvector(1,n);         /* holds original frequency vector */
rec_R = dvector(1,n);         /* holds reconstructed frequency vector */
rec_I = dvector(1,n);         /* holds original frequency vector */
reduce_R = dvector(1,n);      /* holds set of coefficients */
reduce_I = dvector(1,n);      /* holds original frequency vector */
onetimeR = dvector(1,n);      /* holds error for a single vector */
onetimeI = dvector(1,n);      /* holds error for a single vector */
ferr_R = dvector(1,n);        /* holds accumulated error */
ferr_I = dvector(1,n);        /* holds accumulated error */


/* READ TRANSFORM FROM FILE */

/* open Real transform file */
inhandle = fopen(KLT_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_R);
     exit(-1);
}

/* load real KL from transform file */
for (i=1; i<=n; i++)
     for (j=1; j<=n; j++)
     {
          fscanf(inhandle,"%e",&tempfloat);
          KL_R[i][j] = (double) tempfloat;
     }
fclose(inhandle);

/* open Imaginary transform file */
inhandle = fopen(KLT_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_I);
     exit(-1);
}

/* load imaginary KL from transform file */
for (i=1; i<=n; i++)
     for (j=1; j<=n; j++)
     {
          fscanf(inhandle,"%e",&tempfloat);
          KL_I[i][j] = (double) tempfloat;
     }
```

```
fclose(inhandle);



/* READ FREQUENCY AVERAGES FROM FILE */

/* open real averages file */
inhandle = fopen(AVG_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_R);
     exit(-1);
}

/* load averages from averages file */
for (j=1; j<=n; j++)
{
     fscanf(inhandle,"%e",&tempfloat);
     averageR[j] = (double) tempfloat;
}
fclose(inhandle);

/* open imaginary averages file */
inhandle = fopen(AVG_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_I);
     exit(-1);
}

/* load averages from averages file */
for (j=1; j<=n; j++)
{
     fscanf(inhandle,"%e",&tempfloat);
     averageI[j] = (double) tempfloat;
}
fclose(inhandle);



/* KL TRANSFORMATION EXPERIMENT */

/* open temporary file for data */
outhandle = fopen(temp,"w");
if (outhandle == NULL)
{    printf("Can't open file %s.",temp);
     exit(-1);
}
```

```
/* open source file for reading */
inhandle = fopen(argv[1],"r");
if (inhandle == NULL)
{     printf("Can't open file %s.",argv[1]);
      exit(-1);
}

/* count number of vectors in source file */
integers_read = vectors = 0;
while (fscanf(inhandle,"%d",&tempint) != EOF)
{
      integers_read++;
      if (integers_read == N)
      {
            integers_read = 0;
            vectors++;
      }
}
rewind(inhandle);

/* transform the data file */
integers_read = 0;
j = 1;
while (j <= vectors)
{
      fscanf(inhandle,"%d",&tempint);
      integers_read++;
      buff[integers_read] = tempint;

      /* transform a vector */
      if (integers_read == N)
      {
            /* load input array to fourier transform */
            for (k=1; k<=N; k++)
            {
                  d[2*k-1] = (double) buff[k];
                  d[2*k-1] /= MAX_AMP;
            }

            /* overwrite input with complex frequencies */
            four1(d,N,1);

            /* extract real, imaginary and DC components */
            rect(d,n,org_R,org_I,&DC);

            /* subtract mean from each component */
            for (k=1; k<=n; k++)
            {
                  org_R[k] -= averageR[k];
                  org_I[k] -= averageI[k];
            }
```

```
/* generate KL coefficients */
transform__(reduce_R,reduce_I,dim,org_R,org_I,n,KL_R,KL_I);

/* reconstruct frequency vector from KL coefficients */
inverse_transform__(rec_R,rec_I,n,reduce_R,reduce_I,dim,
                    KL_R,KL_I);

/* clear input array for inverse FFT */
for (k=1; k<=2*N; k++)
      d[k] = 0.0;

/* insert DC value */
d[1] = DC;

/* load input array with reconstructed spectrum */
for (k=2; k<=n+1; k++)
{
      d[2*k-1] = rec_R[k-1] + averageR[k-1];
      d[2*k] = rec_I[k-1] + averageI[k-1];
}
for (k=2; k<=n; k++)
{
      d[N+2*k-1] = rec_R[n-k+1] + averageR[n-k+1];
      d[N+2*k] = -1.0 * (rec_I[n-k+1] + averageI[n-k+1]);
}

/* reconstruct time domain data */
four1(d,N,-1);

/* write reconstucted time domain data to output file */
for (k=1; k<=N; k++)
{
      d[2*k-1] *= MAX_AMP;
      d[2*k-1] /= N;
      fprintf(outhandle,"%d\n",(int) d[2*k-1]);
}

/* generate error_spectrum for this vector */
error_spectrum_(n,org_R,org_I,rec_R,rec_I,onetimeR,onetimeI);

/* accumulate error */
for (k=1; k<=n; k++)
{
      ferr_R[k] += onetimeR[k];
      ferr_I[k] += onetimeI[k];
}

/* clear input array for next vector */
for (k=1; k<=2*N; k++)
      d[k] = 0.0;

/* increment vector counter */
j++;
```

```
                /* reset elements per vector counter */
                integers_read=0;
            }
        }
        fclose(inhandle);
        fclose(outhandle);



        /* DETERMINE ERROR FOR EXPERIMENT */

        /* open file for error data */
        outhandle = fopen(error,"a");
        if (outhandle == NULL)
        {     printf("Can't open file %s.");
              exit(-1);
        }

        /* write number of coefficients used this experiment */
        fprintf(outhandle,"%d ",dim);

        /* determine average error per frequency for this file */
        for (k=1; k<=n; k++)
        {
              ferr_R[k] /= vectors;
              ferr_I[k] /= vectors;
        }

        /* find absolute value of real spectrum and write to file */
        magnitude = 0.0;
        for (k=1; k<=n; k++)
              magnitude += ferr_R[k] * ferr_R[k];
        DC = sqrt(magnitude/n);
        fprintf(outhandle, "%e ",DC);

        /* find absolute value of imaginary spectrum and write to file */
        magnitude = 0.0;
        for (k=1; k<=n; k++)
              magnitude += ferr_I[k] * ferr_I[k];
        DC = sqrt(magnitude/n);
        fprintf(outhandle, "%e \n",DC);

        fclose(outhandle);
}
```

APPENDIX E

```
/*                    RELATIVE MEAN SQUARE ERROR PROGRAM

            Command Line Inputs : reduce speech.dat

      Speech source files are first converted from the NeXT sound format into a
 data format using the program sound_to.   Sound_to creates two files, a header
 file which contains the source file's header information and a data file of
 integers between -32768 and 32768.

      This program uses the original sound file's header with the file size
 adjusted to the reconstructed file's size (usually not same size as original).

      The program prompts the user for the number of coefficients used for the
 reduction experiment writes this value to an error file along with the RMSE.


            Outputs : temp_2.dat
                    : error_2.dat


      Program written by FLTLT Don Dryley at AFIT Sep 92.
*/



#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"



#define MAX_AMP 32768
#define N 256
#define n 128



typedef struct
{
      int magic;
      int DataLocation;
      int DataSize;
      int DataFormat;
      int SamplingRate;
      int ChannelCount;
      char info[4];
} SNDheader;
```

```
void transform__(trans_R,trans_I,dim,in_R,in_I,size,KL_R,KL_I)
double *trans_R,*trans_I,*in_R,*in_I,**KL_R,**KL_I;
int dim,size;



/* function transforms two input vectors of dimension size */
/* into two vectors of coefficients of dimension dim using */
/* the transformation matrices KL_R and KL_I               */
{
      int i,j;

      /* transform = KL multiplied by in_vector */
      for (i=1; i<=dim; i++)
      {
            trans_R[i] = trans_I[i] = 0.0;
            for (j=1; j<=size; j++)
            {
                  trans_R[i] += KL_R[i][j] * in_R[j];
                  trans_I[i] += KL_I[i][j] * in_I[j];
            }
      }
}



void inverse_transform__(inv_R,inv_I,size,red_R,red_I,dim,KL_R,KL_I)
double *inv_R,*inv_I,*red_R,*red_I,**KL_R,**KL_I;
int size,dim;
/* function inverse transforms two vectors of coefficients of */
/* dimension dim into two output vectors of dimension size    */
/* using the transformation matrices KL_R and KL_I            */
{
      int i,j;

      /* inverse transform = KLI multiplied by coefficients */
      for (i=1; i<=size; i++)
      {
            inv_R[i] = inv_I[i] = 0.0;
            for (j=1; j<=dim; j++)
            {
                  inv_R[i] += KL_R[j][i] * red_R[j];
                  inv_I[i] += KL_I[j][i] * red_I[j];
            }
      }
}
```

```
void error_spectrum_(size,org_R,org_I,rec_R,rec_I,error_R,error_I)
double *org_R, *org_I, *rec_R, *rec_I, *error_R, *error_I;
int size;
/* function generates a normalised error spectrum by, for each */
/* frequency, dividing the difference between the original and */
/* reconstructed spectrums by the original spectrum.   Assume  */
/* assume that the original, reconstructed, and error vectors  */
/* are created prior to execution with the function vector     */
{
      int i;

      for (i=1; i<=size; i++)
      {
            if (org_R[i] != 0.0)
                  error_R[i] = (org_R[i] - rec_R[i])/org_R[i];
            else if (rec_R[i] != 0.0)
                  error_R[i] = (org_R[i] - rec_R[i])/rec_R[i];
                 else error_R[i] = 0.0;

            if (org_I[i] != 0.0)
                  error_I[i] = (org_I[i] - rec_I[i])/org_I[i];
            else if (rec_I[i] != 0.0)
                  error_I[i] = (org_I[i] - rec_I[i])/rec_I[i];
                 else error_I[i] = 0.0;
      }
}



void main(argc,argv)
int argc;
char *argv[];
{
      SNDheader   SND;
      FILE        *inhandle, *outhandle;
      short       *tempdata;
      int         i, j, k, vectors, integers_read, tempint, dim;
      long        *buff, templong;
      float       tempfloat;
      double      **KL_R, **KL_I, *averageR, *averageI, *d, *org_R, *org_I,
                  *rec_R, *rec_I, *reduce_R, *reduce_I, DC, magnitude, *ferr_R,
                  *ferr_I, *onetimeR, *onetimeI;
      char        KLT_R[] = "KLT_Re.dat",
                  KLT_I[] = "KLT_Im.dat",
                  AVG_R[] = "avgs_Re.dat",
                  AVG_I[] = "avgs_Im.dat",
                  temp[] = "temp_2.dat",
                  error[] = "error_2.dat";
```

```
/* CHECK ARGUMENTS */

if( argc != 2)
{     printf("Format: D>reduce speech.dat ");
      exit(-1);
}



/* PROMPT FOR NUMBER OF KL COEFFICIENTS */

printf("Enter number of coefficients <1 - %d> ... ",n);
scanf("%d",&dim);
printf("\n\n");



/* CREATE MATRICES */

buff = lvector(1,N);          /* input buffer */
d = dvector(1,2*N);           /* FFT of one frame of speech is complex */
averageR = dvector(1,n);      /* average of discrete frequencies */
averageI = dvector(1,n);      /* average of discrete frequencies */
KL_R = dmatrix(1,n,1,n);      /* hold KL transform */
KL_I = dmatrix(1,n,1,n);      /* hold KL transform */
org_R = dvector(1,n);         /* holds original frequency vector */
org_I = dvector(1,n);         /* holds original frequency vector */
rec_R = dvector(1,n);         /* holds reconstructed frequency vector */
rec_I = dvector(1,n);         /* holds original frequency vector */
reduce_R = dvector(1,n);      /* holds set of coefficients */
reduce_I = dvector(1,n);      /* holds original frequency vector */
onetimeR = dvector(1,n);      /* holds error for a single vector */
onetimeI = dvector(1,n);      /* holds error for a single vector */
ferr_R = dvector(1,n);        /* holds accumulated error */
ferr_I = dvector(1,n);        /* holds accumulated error */



/* READ TRANSFORM FROM FILE */

/* open Real transform file */
inhandle = fopen(KLT_R,"r");
if(inhandle == NULL)
{     printf("Can't open file %s." ,KLT_R);
       exit(-1);
}
```

```
/* load real KL from transform file */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(inhandle,"%e",&tempfloat);
            KL_R[i][j] = (double) tempfloat;
      }
fclose(inhandle);

/* open Imaginary transform file */
inhandle = fopen(KLT_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_I);
     exit(-1);
}

/* load imaginary KL from transform file */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(inhandle,"%e",&tempfloat);
            KL_I[i][j] = (double) tempfloat;
      }
fclose(inhandle);



/* READ FREQUENCY AVERAGES FROM FILE */

/* open real averages file */
inhandle = fopen(AVG_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_R);
     exit(-1);
}

/* load averages from averages file */
for (j=1; j<=n; j++)
{
      fscanf(inhandle,"%e",&tempfloat);
      averageR[j] = (double) tempfloat;
}
fclose(inhandle);

/* open imaginary averages file */
inhandle = fopen(AVG_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_I);
     exit(-1);
}
```

```
/* load averages from averages file */
for (j=1; j<=n; j++)
{
      fscanf(inhandle,"%e",&tempfloat);
      averageI[j] = (double) tempfloat;
}
fclose(inhandle);



/* KL TRANSFORMATION EXPERIMENT */

/* open temporary file for data */
outhandle = fopen(temp,"w");
if (outhandle == NULL)
{     printf("Can't open file %s.",temp);
      exit(-1);
}

/* open source file for reading */
inhandle = fopen(argv[1],"r");
if (inhandle == NULL)
{     printf("Can't open file %s.",argv[1]);
      exit(-1);
}

/* count number of vectors in source file */
integers_read = vectors = 0;
while (fscanf(inhandle,"%d",&tempint) != EOF)
{
      integers_read++;
      if (integers_read == N)
      {
            integers_read = 0;
            vectors++;
      }
}
rewind(inhandle);

/* transform the data file */
integers_read = 0;
j = 1;
while (j <= vectors)
{
      fscanf(inhandle,"%d",&tempint);
      integers_read++;
      buff[integers_read] = tempint;
```

```
/* transform a vector */
if (integers_read == N)
{
      /* load input array to fourier transform */
      for (k=1; k<=N; k++)
      {
            d[2*k-1] = (double) buff[k];
            d[2*k-1] /= MAX_AMP;
      }

      /* overwrite input with complex frequencies */
      four1(d,N,1);

      /* extract real, imaginary and DC components */
      rect(d,n,org_R,org_I,&DC);

      /* subtract mean from each component */
      for (k=1; k<=n; k++)
      {
            org_R[k] -= averageR[k];
            org_I[k] -= averageI[k];
      }

      /* generate KL coefficients */
      transform__(reduce_R,reduce_I,dim,org_R,org_I,n,KL_R,KL_I);

      /* reconstruct frequency vector from KL coefficients */
      inverse_transform__(rec_R,rec_I,n,reduce_R,reduce_I,dim,
                          KL_R,KL_I);

      /* clear input array for inverse FFT */
      for (k=1; k<=2*N; k++)
            d[k] = 0.0;

      /* insert DC value */
      d[1] = DC;

      /* load input array with reconstructed spectrum */
      for (k=2; k<=n+1; k++)
      {
            d[2*k-1] = rec_R[k-1] + averageR[k-1];
            d[2*k] = rec_I[k-1] + averageI[k-1];
      }
      for (k=2; k<=n; k++)
      {
            d[N+2*k-1] = rec_R[n-k+1] + averageR[n-k+1];
            d[N+2*k] = -1.0 * (rec_I[n-k+1] + averageI[n-k+1]);
      }

      /* reconstruct time domain data */
      four1(d,N,-1);
```

```
                /* write reconstucted time domain data to output file */
                for (k=1; k<=N; k++)
                {
                        d[2*k-1] *= MAX_AMP;
                        d[2*k-1] /= N;
                        fprintf(outhandle,"%d\n",(int) d[2*k-1]);
                }

                /* generate error_spectrum for this vector */
                error_spectrum_(n,org_R,org_I,rec_R,rec_I,onetimeR,onetimeI);

                /* accumulate error */
                for (k=1; k<=n; k++)
                {
                        ferr_R[k] += onetimeR[k];
                        ferr_I[k] += onetimeI[k];
                }

                /* clear input array for next vector */
                for (k=1; k<=2*N; k++)
                        d[k] = 0.0;

                /* increment vector counter */
                j++;

                /* reset elements per vector counter */
                integers_read=0;
        }
}
fclose(inhandle);
fclose(outhandle);



/* DETERMINE ERROR FOR EXPERIMENT */

/* open file for error data */
outhandle = fopen(error,"a");
if (outhandle == NULL)
{       printf("Can't open file %s.");
        exit(-1);
}

/* write number of coefficients used this experiment */
fprintf(outhandle,"%d ",dim);

/* determine average error per frequency for this file */
for (k=1; k<=n; k++)
{
        ferr_R[k] /= vectors;
        ferr_I[k] /= vectors;
}
```

```
        /* find absolute value of real spectrum and write to file */
        magnitude = 0.0;
        for (k=1; k<=n; k++)
             magnitude += ferr_R[k] * ferr_R[k];
        DC = sqrt(magnitude/n);
        fprintf(outhandle, "%e ",DC);

        /* find absolute value of real spectrum and write to file */
        magnitude = 0.0;
        for (k=1; k<=n; k++)
             magnitude += ferr_I[k] * ferr_I[k];
        DC = sqrt(magnitude/n);
        fprintf(outhandle, "%e \n",DC);

        fclose(outhandle);
}
```

APPENDIX F

```
/*                          RMSE PER COEFFICIENT PROGRAM

            Command Line Inputs : reduce speech.dat

      Speech source files are first converted from the NeXT sound format into a
 data format using the program sound_to.   Sound_to creates two files, a header
 file which contains the source file's header information and a data file of
 integers between -32768 and 32768.

      This program uses the original sound file's header with the file size
 adjusted to the reconstructed file's size (usually not same size as original).

      The program prompts the user for the number of coefficients used for the
 reduction experiment writes this value to an error file along with the RMSE per
 coefficient.


            Outputs : temp_2.dat
                    : error_2.dat


      Program written by FLTLT Don Dryley at AFIT Sep 92.
*/


#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define MAX_AMP 32768
#define N 256
#define n 128


typedef struct
{
      int magic;
      int DataLocation;
      int DataSize;
      int DataFormat;
      int SamplingRate;
      int ChannelCount;
      char info[4];
} SNDheader;
```

```
void transform__(trans_R,trans_I,dim,in_R,in_I,size,KL_R,KL_I)
double *trans_R,*trans_I,*in_R,*in_I,**KL_R,**KL_I;
int dim,size;
/* function transforms two input vectors of dimension size */
/* into two vectors of coefficients of dimension dim using */
/* the transformation matrices KL_R and KL_I               */
{
      int i,j;

      /* transform = KL multiplied by in_vector */
      for (i=1; i<=dim; i++)
      {
            trans_R[i] = trans_I[i] = 0.0;
            for (j=1; j<=size; j++)
            {
                  trans_R[i] += KL_R[i][j] * in_R[j];
                  trans_I[i] += KL_I[i][j] * in_I[j];
            }
      }
}



void inverse_transform__(inv_R,inv_I,size,red_R,red_I,dim,KL_R,KL_I)
double *inv_R,*inv_I,*red_R,*red_I,**KL_R,**KL_I;
int size,dim;
/* function inverse transforms two vectors of coefficients of */
/* dimension dim into two output vectors of dimension size    */
/* using the transformation matrices KL_R and KL_I            */
{
      int i,j;

      /* inverse transform = KLI multiplied by coefficients */
      for (i=1; i<=size; i++)
      {
            inv_R[i] = inv_I[i] = 0.0;
            for (j=1; j<=dim; j++)
            {
                  inv_R[i] += KL_R[j][i] * red_R[j];
                  inv_I[i] += KL_I[j][i] * red_I[j];
            }
      }
}



void error_spectrum_(size,org_R,org_I,rec_R,rec_I,error_R,error_I)
double *org_R, *org_I, *rec_R, *rec_I, *error_R, *error_I;
int size;
/* function generates a normalised error spectrum by, for each */
/* frequency, dividing the difference between the original and */
/* reconstructed spectrums by the original spectrum.   Assume  */
/* assume that the original, reconstructed, and error vectors  */
```

```
/* are created prior to execution with the function vector     */
{
      int i;

      for (i=1; i<=size; i++)
      {
            if (org_R[i] != 0.0)
                  error_R[i] = (org_R[i] - rec_R[i])/org_R[i];
            else if (rec_R[i] != 0.0)
                  error_R[i] = (org_R[i] - rec_R[i])/rec_R[i];
                 else error_R[i] = 0.0;

            if (org_I[i] != 0.0)
                  error_I[i] = (org_I[i] - rec_I[i])/org_I[i];
            else if (rec_I[i] != 0.0)
                  error_I[i] = (org_I[i] - rec_I[i])/rec_I[i];
                 else error_I[i] = 0.0;
      }
}



void main(argc,argv)
int argc;
char *argv[];
{
      SNDheader   SND;
      FILE        *inhandle, *outhandle;
      short       *tempdata;
      int         i, j, k, vectors, integers_read, tempint, dim;
      long        *buff, templong;
      float       tempfloat;
      double      **KL_R, **KL_I, *averageR, *averageI, *d, *org_R, *org_I,
                  *rec_R, *rec_I, *reduce_R, *reduce_I, DC, magnitude, *ferr_R,
                  *ferr_I, *onetimeR, *onetimeI;
      char        KLT_R[] = "KLT_Re.dat",
                  KLT_I[] = "KLT_Im.dat",
                  AVG_R[] = "avgs_Re.dat",
                  AVG_I[] = "avgs_Im.dat",
                  temp[] = "temp_2.dat",
                  error[] = "error_2.dat";



     /* CHECK ARGUMENTS */

     if( argc != 2)
           printf("Format: D>reduce speech.dat ");
           exit(-1);
     }
```

```
/* PROMPT FOR NUMBER OF KL COEFFICIENTS */

printf("Enter number of coefficients <1 - %d> ... ",n);
scanf("%d",&dim);
printf("\n\n");



/* CREATE MATRICES */

buff = lvector(1,N);          /* input buffer */
d = dvector(1,2*N);           /* FFT of one frame of speech is complex */
averageR = dvector(1,n);      /* average of discrete frequencies */
averageI = dvector(1,n);      /* average of discrete frequencies */
KL_R = dmatrix(1,n,1,n);      /* hold KL transform */
KL_I = dmatrix(1,n,1,n);      /* hold KL transform */
org_R = dvector(1,n);         /* holds original frequency vector */
org_I = dvector(1,n);         /* holds original frequency vector */
rec_R = dvector(1,n);         /* holds reconstructed frequency vector */
rec_I = dvector(1,n);         /* holds original frequency vector */
reduce_R = dvector(1,n);      /* holds set of coefficients */
reduce_I = dvector(1,n);      /* holds original frequency vector */
onetimeR = dvector(1,n);      /* holds error for a single vector */
onetimeI = dvector(1,n);      /* holds error for a single vector */
ferr_R = dvector(1,n);        /* holds accumulated error */
ferr_I = dvector(1,n);        /* holds accumulated error */



/* READ TRANSFORM FROM FILE */

/* open Real transform file */
inhandle = fopen(KLT_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_R);
     exit(-1);
}

/* load real KL from transform file */
for (i=1; i<=n; i++)
     for (j=1; j<=n; j++)
     {
          fscanf(inhandle,"%e",&tempfloat);
          KL_R[i][j] = (double) tempfloat;
     }
fclose(inhandle);

/* open Imaginary transform file */
inhandle = fopen(KLT_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_I);
     exit(-1);
}
```

```
/* load imaginary KL from transform file */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(inhandle,"%e",&tempfloat);
            KL_I[i][j] = (double) tempfloat;
      }
fclose(inhandle);



/* READ FREQUENCY AVERAGES FROM FILE */

/* open real averages file */
inhandle = fopen(AVG_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_R);
      exit(-1);
}

/* load averages from averages file */
for (j=1; j<=n; j++)
{
      fscanf(inhandle,"%e",&tempfloat);
      averageR[j] = (double) tempfloat;
}
fclose(inhandle);

/* open imaginary averages file */
inhandle = fopen(AVG_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_I);
      exit(-1);
}

/* load averages from averages file */
for (j=1; j<=n; j++)
{
      fscanf(inhandle,"%e",&tempfloat);
      averageI[j] = (double) tempfloat;
}
fclose(inhandle);



/* KL TRANSFORMATION EXPERIMENT */

/* open temporary file for data */
outhandle = fopen(temp,"w");
if (outhandle == NULL)
{     printf("Can't open file %s.",temp);
      exit(-1);
}
```

```
/* open source file for reading */
inhandle = fopen(argv[1],"r");
if (inhandle == NULL)
{     printf("Can't open file %s.",argv[1]);
      exit(-1);
}

/* count number of vectors in source file */
integers_read = vectors = 0;
while (fscanf(inhandle,"%d",&tempint) != EOF)
{
      integers_read++;
      if (integers_read == N)
      {
            integers_read = 0;
            vectors++;
      }
}
rewind(inhandle);

/* transform the data file */
integers_read = 0;
j = 1;
while (j <= vectors)
{
      fscanf(inhandle,"%d",&tempint);
      integers_read++;
      buff[integers_read] = tempint;

      /* transform a vector */
      if (integers_read == N)
      {
            /* load input array to fourier transform */
            for (k=1; k<=N; k++)
            {
                  d[2*k-1] = (double) buff[k];
                  d[2*k-1] /= MAX_AMP;
            }

            /* overwrite input with complex frequencies */
            four1(d,N,1);

            /* extract real, imaginary and DC components */
            rect(d,n,org_R,org_I,&DC);

            /* subtract mean from each component */
            for (k=1; k<=n; k++)
            {
                  org_R[k] -= averageR[k];
                  org_I[k] -= averageI[k];
            }
```

```
/* generate KL coefficients */
transform__(reduce_R,reduce_I,dim,org_R,org_I,n,KL_R,KL_I);

/* reconstruct frequency vector from KL coefficients */
inverse_transform__(rec_R,rec_I,n,reduce_R,reduce_I,dim,
                    KL_R,KL_I);

/* clear input array for inverse FFT */
for (k=1; k<=2*N; k++)
      d[k] = 0.0;

/* insert DC value */
d[1] = DC;

/* load input array with reconstructed spectrum */
for (k=2; k<=n+1; k++)
{
      d[2*k-1] = rec_R[k-1] + averageR[k-1];
      d[2*k] = rec_I[k-1] + averageI[k-1];
}
for (k=2; k<=n; k++)
{
      d[N+2*k-1] = rec_R[n-k+1] + averageR[n-k+1];
      d[N+2*k] = -1.0 * (rec_I[n-k+1] + averageI[n-k+1]);
}

/* reconstruct time domain data */
fourl(d,N,-1);

/* write reconstucted time domain data to output file */
for (k=1; k<=N; k++)
{
      d[2*k-1] *= MAX_AMP;
      d[2*k-1] /= N;
      fprintf(outhandle,"%d\n",(int) d[2*k-1]);
}

/* generate error_spectrum for this vector */
error_spectrum_(n,org_R,org_I,rec_R,rec_I,onetimeR,onetimeI);

/* accumulate error */
for (k=1; k<=n; k++)
{
      ferr_R[k] += onetimeR[k];
      ferr_I[k] += onetimeI[k];
}

/* clear input array for next vector */
for (k=1; k<=2*N; k++)
      d[k] = 0.0;

/* increment vector counter */
j++;
```

```
                    /* reset elements per vector counter */
                    integers_read=0;
               }
          }
          fclose(inhandle);
          fclose(outhandle);



          /* DETERMINE ERROR FOR EXPERIMENT */

          /* open file for error data */
          outhandle = fopen(error,"a");
          if (outhandle == NULL)
          {     printf("Can't open file %s.");
                exit(-1);
          }

          /* write number of coefficients used this experiment */
          fprintf(outhandle,"%d ",dim);

          /* determine average error per frequency for this file */
          for (k=1; k<=n; k++)
          {
                ferr_R[k] /= vectors;
                ferr_I[k] /= vectors;
          }

          /* find absolute value of real spectrum and write to file */
          magnitude = 0.0;
          for (k=1; k<=n; k++)
                magnitude += ferr_R[k] * ferr_R[k];
          DC = sqrt(magnitude/n)/dim;
          fprintf(outhandle, "%e ",DC);

          /* find absolute value of imaginary spectrum and write to file */
          magnitude = 0.0;
          for (k=1; k<=n; k++)
                magnitude += ferr_I[k] * ferr_I[k];
          DC = sqrt(magnitude/n)/dim;
          fprintf(outhandle, "%e \n",DC);

          fclose(outhandle);
     }
```

APPENDIX G

LISTING ONE

```
/*                      CONDITION NUMBER PROGRAM

            Command Line Inputs : Nil

      The following program decomposes the two covariance matrices into the
product of three matrices V.W.V^T.  The rows of the KL transform matrices are
formed from the columns of V (eigenvectors) and the singular values (eigenvalues
squared) are contained in the diagonal matrix W.

      The singular values are ranked in descending order.  The ratio of the
largest SV to the nth SV is trhe condition number.  Pairs of condition
numbers for each decomposed covariance matrix are averaged and written to a
file.

            OUTPUTS : File KLT_Re.dat
                    : File KLT_Im.dat
                    : accum_Im.dat

      Program written by FLTLT Don Dryley at AFIT, Sep 1992
*/


#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define MAX_AMP 32768
#define N 256
#define n 128


void main(argc,argv)
int argc;
char *argv[];
{
      FILE        *inhandle,*outlhandle,*out2handle;
      int         i, j, k, x, vectors, integers_read, tempint;
      long        *buff;
```

```
float       tempfloat;
double      *d, *data_R, *data_I, *averageR, *averageI, **A_R, **A_I, *W,
            *S, **V, **Cov, DC, temp, zero = 0.0;
char        covarianceR[] = "covar_Re.dat",
            covarianceI[] = "covar_Im.dat",
            transformR[] = "KLT_Re.dat",
            transformI[] = "KLT_Im.dat",
            eigenvalR[] = "eig_Re.dat",
            eigenvalI[] = "eig_Im.dat",
            accumI[] = "accum_Im.dat";


/* CREATE ARRAYS, INITIALISED TO ZERO */

buff = lvector(1,N);        /* input buffer */
d = dvector(1,2*N);         /* FFT of speech data is complex */
data_R = dvector(1,n);      /* Single vector of real coomponents */
data_I = dvector(1,n);      /* Single vector of imaginary components */
averageR = dvector(1,n);    /* average of real components */
averageI = dvector(1,n);    /* average of imaginary components */
A_R = dmatrix(1,n,1,n);     /* covariance matrix of real components */
A_I = dmatrix(1,n,1,n);     /* covariance matrix of imaginary components */
V = dmatrix(1,n,1,n);       /* Matrix of eigenvectors */
W = dvector(1,n);           /* Matrix of singular values */
S = dvector(1,n);


/* LOAD COVARIANCE MATRICES */

/* open Real covariance file */
outlhandle = fopen(covarianceR,"r");
if (outlhandle == NULL)
{
      printf("Can't open file %s, Exiting to system\n",covarianceR);
      exit(-1);
}

/* load covariance array */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(outlhandle,"%e",&tempfloat);
            A_R[i][j] = (double) tempfloat;
      }
fclose(outlhandle);
```

```
/* open Imaginary covariance file */
out2handle = fopen(covarianceI,"r");
if (out2handle == NULL)
{
      printf("Can't open file %s, Exiting to system\n",covarianceI);
      exit(-1);
}

/* load covariance array */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(out2handle,"%e",&tempfloat);
            A_I[i][j] = (double) tempfloat;
      }
fclose(out2handle);



/* FORM KL TRANSFORM OF REAL VECTORS */

/* find and sort eigenvectors of A_R and A_I */
svdcmp(A_R,n,n,W,V);
eigsrt(W,A_R,n);

/* open KL_R transform file */
outlhandle = fopen (transformR,"w");
if(outlhandle == NULL)
{     printf("Can't open file %s",transformR);
      exit(-1);
}

/* write eigenvectors to KL_R transform file */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
            fprintf(outlhandle,"%e\n",(float) A_R[j][i]);
fclose(outlhandle);



/* FORM KL TRANSFORM OF IMAGINARY VECTORS */

/* decompose matrix and rank eigenvalues/vectors */
svdcmp(A_I,n,n,S,V);
eigsrt(S,A_I,n);

/* open KL_I transform file */
outlhandle = fopen (transformI,"w");
if(outlhandle == NULL)
{     printf("Can't open file %s",transformI);
      exit(-1);
}
```

```
        /* write imaginary parts of eigenvectors to KL_I transform file */
        for (i=1; i<=n; i++)
              for (j=1; j<=n; j++)
                    fprintf(outlhandle,"%e\n",(float) A_I[j][i]);
        fclose(outlhandle);

        /* open accumulation file */
        out2handle = fopen (accumI,"w");
        if(out2handle == NULL)
        {     printf("Can't open file %s",accumI);
              exit(-1);
        }

        /* write condition number to file */
        for (i=1; i<=n; i++)
              fprintf(out2handle,"%e\n",(float) ((W[1]/W[i])+(S[1]/S[i]))/2);
        fclose(out2handle);
        fclose(outlhandle);
}
```

LISTING TWO

```
/*                          ENERGY RATIO PROGRAM

          Command Line Inputs : filename of source speech.dat

     Speech source files are first converted from the NeXT sound format into a
data format using the program sound_to.   Sound_to creates two files, a header
file which contains the source file's header information and a data file of
integers which between -32768 and 32768.

     This program uses the original sound file's header with the file size
adjusted to the reconstructed file size (usually not same size as original).

     The program prompts the user for the number of coefficients used for the
reduction experiment and reads the energy represented by that number of
coefficients.

          Outputs : temp_2.dat
                  : error_2.dat

     Program written by FLTLT Don Dryley at AFIT Sep 92
*/


#include <fcntl.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "recipes.h"


#define MAX_AMP 32768
#define N 256
#define n 128


typedef struct
{
     int magic;
     int DataLocation;
     int DataSize;
     int DataFormat;
```

```
        int SamplingRate;
        int ChannelCount;
        char info[4];
} SNDheader;

void transform__(trans_R,trans_I,dim,in_R,in_I,size,KL_R,KL_I)
double *trans_R,*trans_I,*in_R,*in_I,**KL_R,**KL_I;
int dim,size;
/* function transforms two input vectors of dimension size */
/* into two vectors of coefficients of dimension dim using */
/* the transformation matrices KL_R and KL_I               */
{
      int i,j;

      /* transform = KL multiplied by in_vector */
      for (i=1; i<=dim; i++)
      {
            trans_R[i] = trans_I[i] = 0.0;
            for (j=1; j<=size; j++)
            {
                  trans_R[i] += KL_R[i][j] * in_R[j];
                  trans_I[i] += KL_I[i][j] * in_I[j];
            }
      }
}



void inverse_transform__(inv_R,inv_I,size,red_R,red_I,dim,KL_R,KL_I)
double *inv_R,*inv_I,*red_R,*red_I,**KL_R,**KL_I;
int size,dim;
/* function inverse transforms two vectors of coefficients of */
/* dimension dim into two output vectors of dimension size    */
/* using the transformation matrices KL_R and KL_I            */
{
      int i,j;

      /* inverse transform = KLI multiplied by coefficients */
      for (i=1; i<=size; i++)
      {
            inv_R[i] = inv_I[i] = 0.0;
            for (j=1; j<=dim; j++)
            {
                  inv_R[i] += KL_R[j][i] * red_R[j];
                  inv_I[i] += KL_I[j][i] * red_I[j];
            }
      }
}
```

```
void error_spectrum_(size,org_R,org_I,rec_R,rec_I,error_R,error_I)
double *org_R, *org_I, *rec_R, *rec_I, *error_R, *error_I;
int size;
/* function generates a normalised error spectrum by, for each */
/* frequency, dividing the difference between the original and */
/* reconstructed spectrums by the original spectrum.   Assume  */
/* assume that the original, reconstructed, and error vectors  */
/* are created prior to execution with the function vector     */
{
      int i;

      for (i=1; i<=size; i++)
      {
            if (org_R[i] != 0.0)
                  error_R[i] = (org_R[i] - rec_R[i])/org_R[i];
            else if (rec_R[i] != 0.0)
                  error_R[i] = (org_R[i] - rec_R[i])/rec_R[i];
                 else error_R[i] = 0.0;

            if (org_I[i] != 0.0)
                  error_I[i] = (org_I[i] - rec_I[i])/org_I[i];
            else if (rec_I[i] != 0.0)
                  error_I[i] = (org_I[i] - rec_I[i])/rec_I[i];
                 else error_I[i] = 0.0;
      }
}



void main(argc,argv)
int argc;
char *argv[];
{
      SNDheader   SND;
      FILE        *inhandle, *outhandle;
      short       *tempdata;
      int         i, j, k, vectors, integers_read, tempint, dim;
      long        *buff, templong;
      float       tempfloat;
      double      **KL_R, **KL_I, *averageR, *averageI, *d, *org_R, *org_I,
                  *rec_R, *rec_I,   *reduce_R, *reduce_I, DC, mag_R, mag_I,
                  *ferr_R, *ferr_I, *onetimeR, *onetimeI;
      char        KLT_R[] = "KLT_Re.dat",
                  KLT_I[] = "KLT_Im.dat",
                  AVG_R[] = "avgs_Re.dat",
                  AVG_I[] = "avgs_Im.dat",
                  temp[] = "temp_2.dat",
                  error[] = "error_2.dat",
                  eigenvalR[] = "eig_Re.dat",
                  eigenvalI[] = "eig_Im.dat";
```

```
/* CHECK ARGUMENTS */

if( argc != 2)
{     printf("Format: D>reduce speech.dat ");
      exit(-1);
}



/* PROMPT FOR NUMBER OF KL COEFFICIENTS */

printf("Enter number of coefficients <1 - %d> ... ",n);
scanf("%d",&dim);
printf("\n\n");

/* open Real eigenvalues file */
inhandle = fopen(eigenvalR,"r");
if(inhandle == NULL)
{     printf("Can't open file %s." ,eigenvalR);
      exit(-1);
}

/* accumulate first dim coefficients */
mag_R = 0.0;
for (i=1; i<=dim; i++)
{
      fscanf(inhandle,"%e",&tempfloat);
      magnitude += (double) tempfloat;
}
fclose(inhandle);


/* open Imaginary eigenvalues file */
inhandle = fopen(eigenvalI,"r");
if(inhandle == NULL)
{     printf("Can't open file %s." ,eigenvalI);
      exit(-1);
}

/* accumulate first forty coefficients */
mag_I = 0.0;
for (i=1; i<=dim; i++)
{
      fscanf(inhandle,"%e",&tempfloat);
      mag_I += (double) tempfloat;
}
fclose(inhandle);
```

```
/* CREATE MATRICES */

buff = lvector(1,N);        /* input buffer */
d = dvector(1,2*N);         /* FFT of speech data is complex */
averageR = dvector(1,n);    /* average of discrete frequencies */
averageI = dvector(1,n);    /* average of discrete frequencies */
KL_R = dmatrix(1,n,1,n);    /* hold KL transform */
KL_I = dmatrix(1,n,1,n);    /* hold KL transform */
org_R = dvector(1,n);       /* holds original frequency vector */
org_I = dvector(1,n);       /* holds original frequency vector */
rec_R = dvector(1,n);       /* holds reconstructed frequency vector */
rec_I = dvector(1,n);       /* holds original frequency vector */
reduce_R = dvector(1,n);    /* holds set of coefficients */
reduce_I = dvector(1,n);    /* holds original frequency vector */
onetimeR = dvector(1,n);    /* holds error spectrum for a single vector */
onetimeI = dvector(1,n);    /* holds error spectrum for a single vector */
ferr_R = dvector(1,n);      /* holds accumulated error over all vectors */
ferr_I = dvector(1,n);      /* holds accumulated error over all vectors */


/* READ TRANSFORM FROM FILE */

/* open Real transform file */
inhandle = fopen(KLT_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_R);
     exit(-1);
}

/* load KL from transform file */
for (i=1; i<=n; i++)
     for (j=1; j<=n; j++)
     {
          fscanf(inhandle,"%e",&tempfloat);
          KL_R[i][j] = (double) tempfloat;
     }
fclose(inhandle);

/* open Imaginary transform file */
inhandle = fopen(KLT_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,KLT_I);
     exit(-1);
}
```

```
/* load KL from transform file */
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
      {
            fscanf(inhandle,"%e",&tempfloat);
            KL_I[i][j] = (double) tempfloat;
      }
fclose(inhandle);



/* READ FREQUENCY AVERAGES FROM FILE */

/* open real averages file */
inhandle = fopen(AVG_R,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_R);
      exit(-1);
}

/* load averages from averages file */
for (j=1; j<=n; j++)
{
      fscanf(inhandle,"%e",&tempfloat);
      averageR[j] = (double) tempfloat;
}
fclose(inhandle);

/* open imaginary averages file */
inhandle = fopen(AVG_I,"r");
if(inhandle == NULL)
{    printf("Can't open file %s." ,AVG_I);
      exit(-1);
}

/* load averages from averages file */
for (j=1; j<=n; j++)
{
      fscanf(inhandle,"%e",&tempfloat);
      averageI[j] = (double) tempfloat;
}
fclose(inhandle);



/* KL TRANSFORMATION EXPERIMENT */

/* open temporary file for data */
outhandle = fopen(temp,"w");
if (outhandle == NULL)
{     printf("Can't open file %s.",temp);
      exit(-1);
}
```

```
/* open source file for reading */
inhandle = fopen(argv[1],"r");
if (inhandle == NULL)
{     printf("Can't open file %s.",argv[1]);
      exit(-1);
}

/* count number of vectors in source file */
integers_read = vectors = 0;
while (fscanf(inhandle,"%d",&tempint) != EOF)
{
      integers_read++;
      if (integers_read == N)
      {
            integers_read = 0;
            vectors++;
      }
}
rewind(inhandle);

/* transform the data file */
integers_read = 0;
j = 1;
while (j <= vectors)
{
      fscanf(inhandle,"%d",&tempint);
      integers_read++;
      buff[integers_read] = tempint;

      /* transform a vector */
      if (integers_read == N)
      {
            /* load input array to fourier transform */
            for (k=1; k<=N; k++)
            {
                  d[2*k-1] = (double) buff[k];
                  d[2*k-1] /= MAX_AMP;
            }

            /* overwrite input with complex frequencies */
            fourl(d,N,1);

            /* extract real, imaginary and DC components */
            rect(d,n,org_R,org_I,&DC);

            /* subtract mean from each component */
            for (k=1; k<=n; k++)
            {
                  org_R[k] -= averageR[k];
                  org_I[k] -= averageI[k];
            }
```

```
                    /* generate KL coefficients */
                    transform__(reduce_R,reduce_I,dim,org_R,org_I,n,KL_R,KL_I);

                    /* reconstruct frequency vector from KL coefficients */
                    inverse_transform__(rec_R,rec_I,n,reduce_R,reduce_I,dim,
                                        KL_R,KL_I);

                    /* clear input array for inverse FFT */
                    for (k=1; k<=2*N; k++)
                          d[k] = 0.0;

                    /* insert DC value */
                    d[1] = DC;

                    /* load input array with reconstructed spectrum */
                    for (k=2; k<=n+1; k++)
                    {
                          d[2*k-1] = rec_R[k-1] + averageR[k-1];
                          d[2*k] = rec_I[k-1] + averageI[k-1];
                    }
                    for (k=2; k<=n; k++)
                    {
                          d[N+2*k-1] = rec_R[n-k+1] + averageR[n-k+1];
                          d[N+2*k] = -1.0 * (rec_I[n-k+1] + averageI[n-k+1]);
                    }

                    /* reconstruct time domain data */
                    four1(d,N,-1);

                    /* write reconstucted time domain data to output file */
                    for (k=1; k<=N; k++)
                    {
                          d[2*k-1] *= MAX_AMP;
                          d[2*k-1] /= N;
                          fprintf(outhandle,"%d\n",(int) d[2*k-1]);
                    }
                    /* clear input array for next vector */
                    for (k=1; k<=2*N; k++)
                          d[k] = 0.0;

                    /* increment vector counter */
                    j++;

                    /* reset elements per vector counter */
                    integers_read=0;
              }
        }
        fclose(inhandle);
        fclose(outhandle);
```

```
        /* DETERMINE ERROR FOR EXPERIMENT */

        /* open file for error data */
        outhandle = fopen(error,"a");
        if (outhandle == NULL)
        {     printf("Can't open file %s.");
              exit(-1);
        }

        /* write number of coefficients used this experiment */
        fprintf(outhandle,"%d ",dim);

        /* write energy ratios for real amd imaginary coefficients */
        fprintf(outhandle, "%e %e",mag_R, mag_I);

        fclose(outhandle);
}
```

APPENDIX H

TRAINING SET

There comes sly shoe.

That blue shack.

Then move but soon.

She shock beans.

Black peep slew.

The slack moon.

TESTING SET

Fly sheep beams.